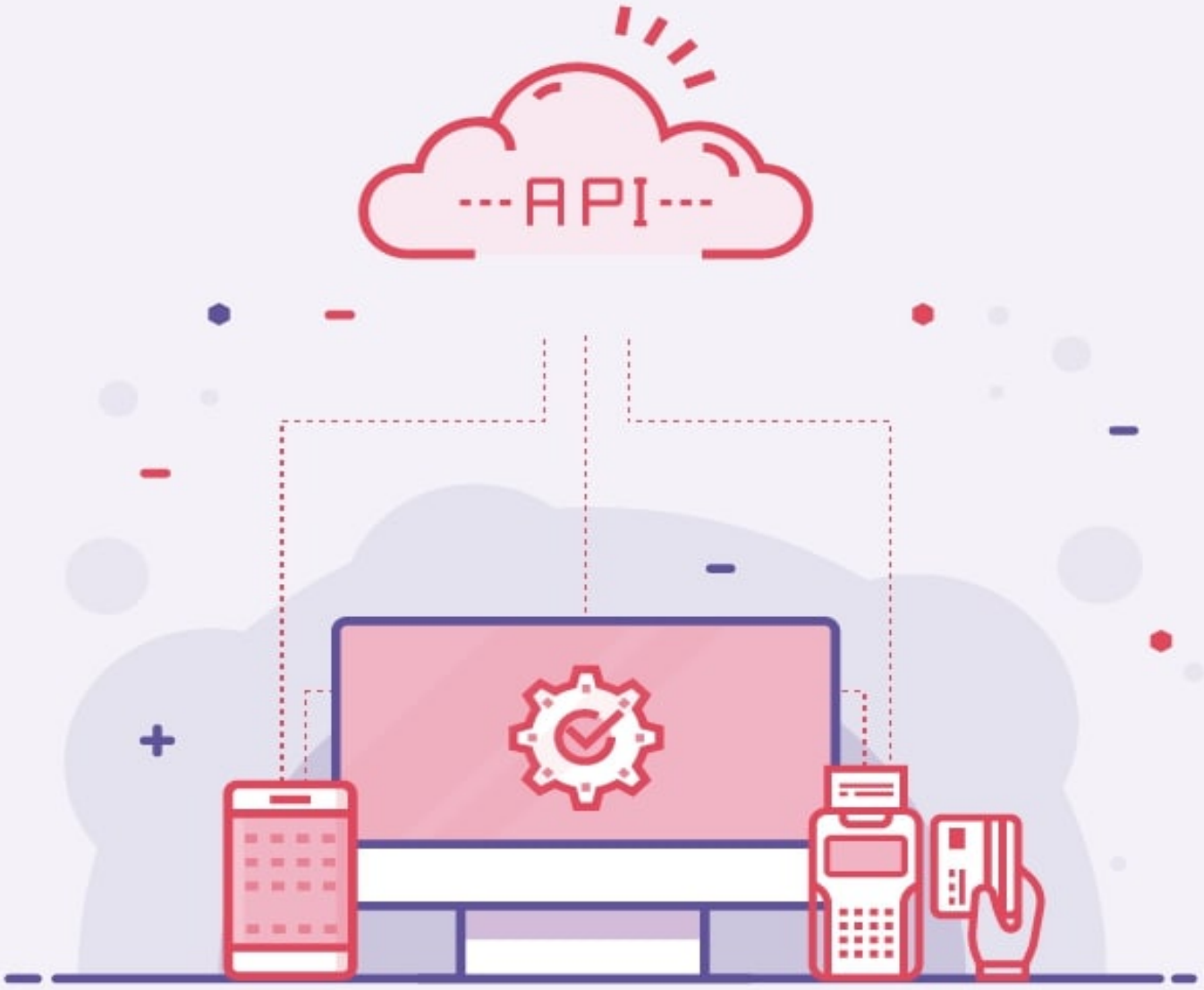
APIs REST em Kotlin

Seus serviços prontos para o mundo real



Casa do Código

ALEXANDRE SAUDATE

Sumário

- ISBN
- Agradecimentos
- Sobre o autor
- Pré-requisitos
- Introdução
- Serviços em REST
  - 1 O que é REST, afinal?
  - 2 Expandindo o nosso serviço inicial
  - 3 Criando relacionamentos entre recursos
  - 4 Criando clientes REST
  - 5 Criando os testes automatizados
  - 6 Segurança
- APIs
  - 7 APIs
  - 8 Documentando a API
  - 9 Outras técnicas
  - 10 Considerações finais
  - 11 Referências bibliográficas

ISBN

Impresso: 978-65-86110-71-5

Digital: 978-65-86110-73-9

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Muitas pessoas fizeram parte da concepção deste livro, participando direta ou indiretamente. Este, em particular, é uma reescrita do meu livro *APIs REST: Seus serviços prontos para o mundo real* - que por sua vez, nasceu como segunda versão do meu livro *REST: Construa APIs inteligentes de maneira simples* (tendo sido dissociado posteriormente). Isso significa que existe uma história de no mínimo sete anos envolvida.

Gostaria de agradecer primeiramente a Deus. Sem Ele, nada seria possível.

Gostaria também de agradecer ao meu amigo Aldrin Leal. Conheci o Aldrin trabalhando com ele e, desde então, toda vez que fazemos algo juntos o resultado é fantástico. Muitas vezes, quando tenho dúvidas sobre alguma coisa, recorro a ele para me iluminar.

Gostaria também de agradecer ao Erico Boskovisch Madeira, que foi o leitor beta do livro anterior e fez várias considerações antes do seu lançamento. Muitas das sugestões foram incorporadas a este livro, tornando-o um trabalho melhor.

Gostaria também de agradecer a todos que compraram o meu livro de REST, já mencionado. Muitas técnicas que usei neste novo livro são resultados de *feedbacks* dos leitores do primeiro. Sem dúvida alguma, o suporte do público é fundamental para que nós, autores, tenhamos disposição para escrever e disseminar o conhecimento acumulado nestes anos de estrada.

Sobre o autor

Este é meu quarto livro escrito pela Casa do Código. Antes dele, escrevi *SOA Aplicado: integrando com web services e além*, *REST: Construa APIs inteligentes de maneira simples* e *APIs REST: Seus serviços prontos para o mundo real*.

Trabalho como especialista de software no PagSeguro e anteriormente passei por empresas, como Guichê Virtual, Sciensa e Serasa Experian. Como consultor, prestei serviços a empresas dos mais diversos segmentos de atuação, como Netshoes, CVC, Magazine Luiza, iG, Oi internet, e muitas outras.

Por muitos anos, trabalhei como especialista em SOA. Desde o início procurei sempre ter em mente a concepção original dessa arquitetura - algo muito próximo ao que hoje reconhecemos como microsserviços. Essa experiência me levou a adotar REST ainda em seus primórdios no país. Utilizei REST pela primeira vez em um projeto para o iG em 2011. Desde então, venho estudando cada vez mais sobre o assunto e procurando entender a forma como essa técnica ajuda desenvolvedores e desenvolvedoras em todo o mundo a oferecer aplicações cada vez mais interconectadas.

Pré-requisitos

Este livro é uma história contada sobre uma aplicação fictícia chamada `C.A.R.`, um aplicativo para solicitação de carros para viagens curtas. Essa aplicação é desenvolvida em Kotlin utilizando o Spring Boot como framework para basear tudo e Gradle como controlador de build.

Essas ferramentas foram escolhidas para que você esteja o mais imerso possível no mundo Kotlin. Até o momento da escrita deste livro, Kotlin ainda é percebido pelo mercado como uma linguagem focada no desenvolvimento de aplicativos para Android, e não para aplicações de *backend*.

Assim sendo, o ferramental que utilizaremos é o que proporciona maior imersão na forma de desenvolver aplicações deste tipo utilizando Kotlin.

A **simplicidade** foi o principal conceito utilizado para escrita deste livro. Todos os exemplos foram pensados de forma que você consiga acompanhá-los ainda que não tenha conhecimento sobre Spring Boot ou mesmo sobre programação em Java.

Mas o ideal é que você reescreva estes exemplos em sua máquina, utilizando a IDE IntelliJ IDEA (https://www.jetbrains.com/pt-br/idea/). Esta IDE é criada e mantida pela mesma organização que criou a linguagem Kotlin, a JetBrains.

Tenho certeza de que, dessa forma, você conseguirá tirar o máximo proveito do que apresento aqui.

Introdução

Este livro nasceu como uma releitura do meu livro recém-publicado *APIs REST: Seus serviços prontos para o mundo real*. Tive a ideia de reescrevê-lo em Kotlin pois, no momento, Kotlin ainda é vista pelo mercado como uma linguagem extremamente promissora, porém mais fortemente relegada ao nicho de desenvolvimento de aplicações para Android.

A proposta deste livro, portanto, é abordar os mesmos exemplos utilizados no livro de APIs REST, porém em Kotlin. Se você já possui o outro livro, acredito que seja especialmente interessante ler este para que você possa fazer o comparativo entre as linguagens. Kotlin é uma linguagem que permite escrita de códigos tanto sob o paradigma orientado a objetos quanto sob o paradigma funcional, e mesmo assim, o faz de maneira harmônica entre ambos. É possível escrever extensões de classes fora destas e também DSLs (os próprios arquivos Gradle utilizados na escrita deste livro estão em Kotlin). Em resumo, é uma linguagem extremamente poderosa, que permite que criemos aplicações de fácil leitura e manutenção, além de se beneficiar da execução sobre a JVM.

Este livro, assim como sua contrapartida em Java, é dividido em duas partes: a primeira, voltada à parte mais básica sobre serviços REST e por onde iniciar esse desenvolvimento. A segunda parte, sobre APIs, procura abordar as questões necessárias para que os serviços REST sejam realmente aproveitados por outros desenvolvedores. Em outras palavras, como podemos criar APIs expostas para o público.

Assim sendo, este livro foi dividido nos seguintes capítulos:

O capítulo 1 apresenta o aplicativo `C.A.R.`, que faz intermediação entre pessoas que precisam de um transporte rápido e motoristas que possuem um carro e podem prestar esse serviço. Neste capítulo, criaremos a primeira API do aplicativo, que recupera dados de passageiros. Apresento alguns conceitos básicos, como os

principais métodos HTTP necessários para desenvolver uma API REST.

No capítulo 2, essa API será expandida e veremos como recuperar dados de passageiros específicos, além de como criar esses dados, atualizá-los (total ou parcial) e apagá-los - tudo isso com um cliente especializado, o `Postman`. Também veremos como funciona um dos pilares fundamentais de REST, o da negociação de conteúdo, e conheceremos a idempotência e seus efeitos práticos sobre as APIs.

No capítulo 3, nós vamos criar a API de solicitação de viagens e vamos relacionar essa API com a de gerenciamento de passageiros através do uso de HATEOAS (uma das técnicas mais importantes de REST).

No capítulo 4, utilizaremos o serviço do Google Maps para incrementar a API criada no capítulo anterior. Desta forma, veremos como criar um cliente de uma API REST e também vamos conhecer o JSONPath, uma poderosa técnica de busca de dados dentro de um documento JSON.

No capítulo 5, vamos realizar os testes de ponta a ponta. Vamos aprender como criar um dublê do serviço do Google Maps com o `Wiremock` (desta forma, dispensando o consumo da API quando estivermos executando os testes) e também vamos aprender a usar uma ferramenta especializada em testes de serviços REST, o `REST Assured`.

No capítulo 6, acrescentaremos uma camada de segurança aos nossos serviços. Vamos ver como funciona o HTTPS e vamos gerar um certificado e implantá-lo na nossa API. Também vamos implantar autenticação BASIC nas nossas APIs.

A partir do capítulo 7, inicia-se a segunda parte do livro, sobre APIs. Esse capítulo vai utilizar os conhecimentos já adquiridos ao longo do livro e sedimentá-los. Então, veremos como criar URLs mais significativas para nossos clientes, quais os códigos HTTP mais

importantes, como fornecer mensagens de erro significativas, como internacionalizar as mensagens de erro e, finalmente, como criar APIs retrocompatíveis (técnica muito importante em casos de APIs públicas ou aplicativos *mobile*).

No capítulo 8, nós vamos aprender a documentar nossas APIs. Vamos criar uma API viva com `Swagger` e também veremos como usar o Documenter do `Postman`.

No capítulo 9, veremos algumas técnicas que não puderam ser abordadas ao longo do livro. Veremos como paginar serviços, implementar a técnica CORS (importante para consumir os serviços REST a partir de páginas web), autenticação com OAuth e, finalmente, como utilizar o `AWS API Gateway` para que possamos ter mais segurança ao expor APIs públicas. Apresentarei como criar e utilizar sua conta na AWS de forma controlada, para que não haja problemas em relação aos custos.

No capítulo 10, apresento algumas tecnologias que não estão necessariamente relacionadas às APIs em REST, mas fazem parte de um ecossistema mais amplo de APIs.

Este livro foi escrito com muito carinho, levando em consideração a bagagem que acumulei ao longo dos anos e também como resultado de muitas conversas com desenvolvedores e desenvolvedoras como você. Dessa forma, procurei abordar os conceitos que considero os mais importantes.

Caso você, leitor ou leitora, sinta que algum conceito não está suficientemente claro, ou tenha sentido falta de alguma técnica, sinta-se à vontade para entrar em contato comigo diretamente, através do e-mail alesaudate@gmail.com. Esse é meu compromisso com você.

Serviços em REST

Capítulo 1
O que é REST, afinal?

"Quem conduz e arrasta o mundo não são as máquinas, são as ideias." – Victor Hugo.

Hoje em dia, muito se ouve falar de REST e você, como um desenvolvedor ou uma desenvolvedora de software, já deve ter ouvido falar desse termo. Trata-se de um meio de realizar a *comunicação entre dois sistemas diferentes*, independentemente da linguagem em que tenham sido escritos.

Obviamente, existem muitas formas de se fazer isso. Mas o fato é que REST estabelece um conjunto de padrões que permite fazer isso de forma **eficiente** e **interoperável**, coisas que, considerando cenários de microsserviços, por exemplo, se tornam especialmente interessantes de se possuir.

> **O que são microsserviços?**
>
> Segundo Martin Fowler, "[...] o estilo de arquitetura de microsserviços é uma abordagem que desenvolve um aplicativo único, como uma suíte de pequenos serviços, cada um executando seu próprio processo e se comunicando através de mecanismos leves, muitas vezes em uma API com recursos HTTP. Esses serviços são construídos em torno de capacidades de negócios e funcionam através de mecanismos de deploy independentes totalmente automatizados. Há o mínimo possível de gerenciamento centralizado desses serviços, que podem ser escritos em diferentes linguagens de programação e utilizam diferentes tecnologias de armazenamento de dados." (LEWIS; FOWLER, 2015)

Analisemos um *case*: considere que você é o principal desenvolvedor do back-end de um aplicativo para solicitar transporte de carros (em modelo semelhante ao de táxis, como o daquela empresa que começa com U, sabe?). Esse aplicativo se chama **C.A.R.** e sua missão é promover o melhor meio de transporte possível pelo menor preço possível conectando pessoas que possuem carros e gostariam de usá-los para obter uma renda extra com pessoas que precisam de um meio de transporte.

Você está encarregado de criar o back-end, enquanto outros desenvolvedores construirão o front-end (tanto em versão web quanto mobile).

1.1 Utilizando o Spring Boot para criar uma primeira API

Uma das maneiras mais rápidas de se iniciar um projeto de uma API REST em Kotlin hoje em dia é através do uso de algum framework. Por sua relação com Java, é mais conveniente utilizar o framework Java mais popular e mais antigo, o Spring. Ele provê diversas capacidades para o projeto, entre elas a disponibilização e o gerenciamento de ferramentas para construção de APIs. Além disso, o Spring tem um subprojeto chamado Spring Boot, cujo objetivo é tornar a utilização do Spring tão rápida e prática quanto possível.

O Spring também conta com um site chamado Spring Initializr (https://start.spring.io/). Nele, é possível selecionar quais capacidades vamos querer no projeto, quais outros possíveis frameworks Java vamos querer utilizar etc.

OBSERVAÇÃO SOBRE O SPRING INITIALIZR

Sabemos que o mercado de desenvolvimento de softwares é muito dinâmico, por isso muitas plataformas, ferramentas e sites deixam de existir ou são substituídos com frequência. Não há como dizer se isso terá acontecido com o Spring Initializr quando você estiver lendo este livro. No entanto, como mencionado, o Spring é um dos frameworks mais antigos e mais bem estabelecidos do Java, e tudo indica que sua existência vai permanecer enquanto houver Java - logo, provavelmente o Spring Initializr vai continuar existindo ou existirá algo melhor no futuro.

Ao acessar o site, você deve se deparar com uma tela semelhante à seguinte:

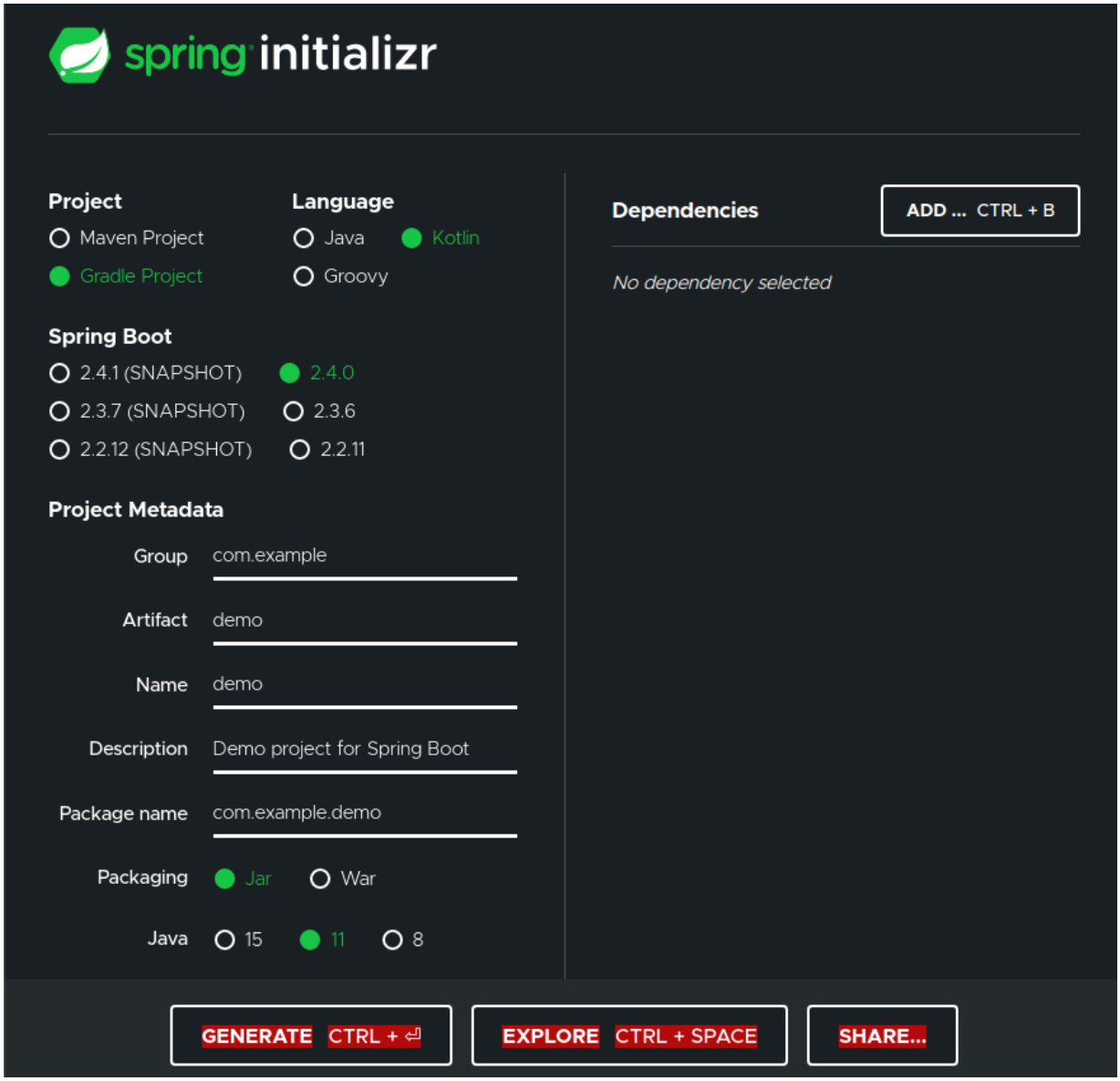


Figura 1.1: Tela do Spring Initializr

Aqui, você tem a opção de selecionar o sistema de build para o seu projeto (Maven ou Gradle), a linguagem (Java, Kotlin ou Groovy), a versão do Spring Boot, alguns metadados sobre o projeto e as dependências. Para manter a consistência, vou utilizar ao longo deste livro, em todos os projetos, o Gradle com linguagem Kotlin, Java 11 e Spring Boot versão 2.4.0. Os metadados devem variar de capítulo para capítulo - caso queira conferir exatamente quais serão, basta verificar o código-fonte do projeto no GitHub (disponível em

https://github.com/alesaudate/rest-kotlin). Finalmente, as dependências também devem variar de capítulo para capítulo por conta das necessidades, então sinalizarei essas mudanças.

Para este capítulo, precisamos utilizar o Spring Web, o Spring Data JPA e o driver do H2 como dependências. Ao começar a digitar web e data, o sistema completará o restante das palavras e a listagem de dependências aparecerá. Ela deve ficar semelhante à seguinte:

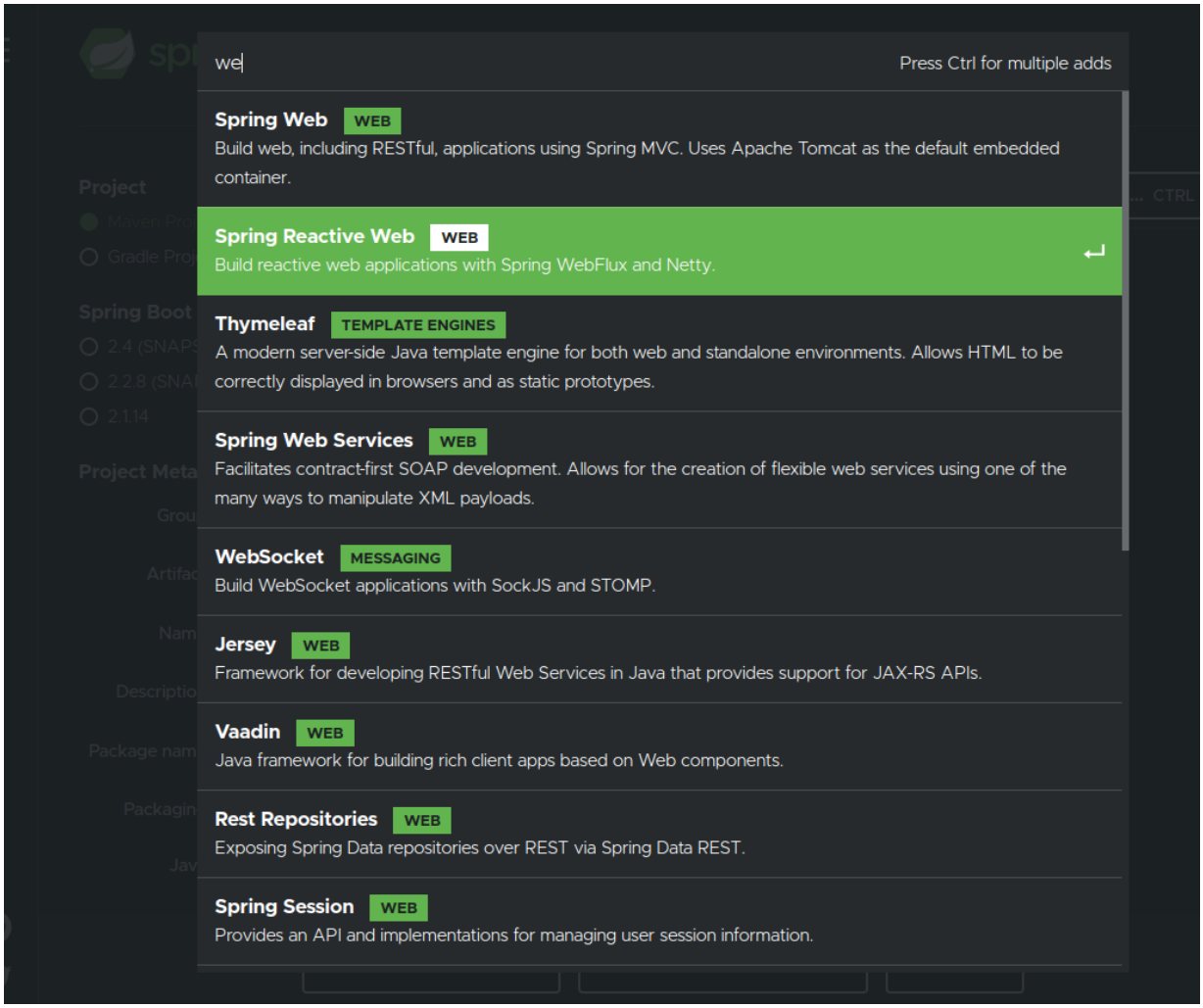

we
Press Ctrl for multiple adds
Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Spring Reactive Web WEB
Build reactive web applications with Spring WebFlux and Netty.
Thymeleaf TEMPLATE ENGINES
A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.
Spring Web Services WEB
Facilitates contract-first SOAP development. Allows for the creation of flexible web services using one of the many ways to manipulate XML payloads.
WebSocket MESSAGING
Build WebSocket applications with SockJS and STOMP.
Jersey WEB
Framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs.
Vaadin WEB
Java framework for building rich client apps based on Web components.
Rest Repositories WEB
Exposing Spring Data repositories over REST via Spring Data REST.
Spring Session WEB
Provides an API and implementations for managing user session information.

Figura 1.2: Tela das dependências do Spring

Por fim, aperte o botão `Generate` e o download de um arquivo `zip` deve ser iniciado com o seguinte conteúdo:

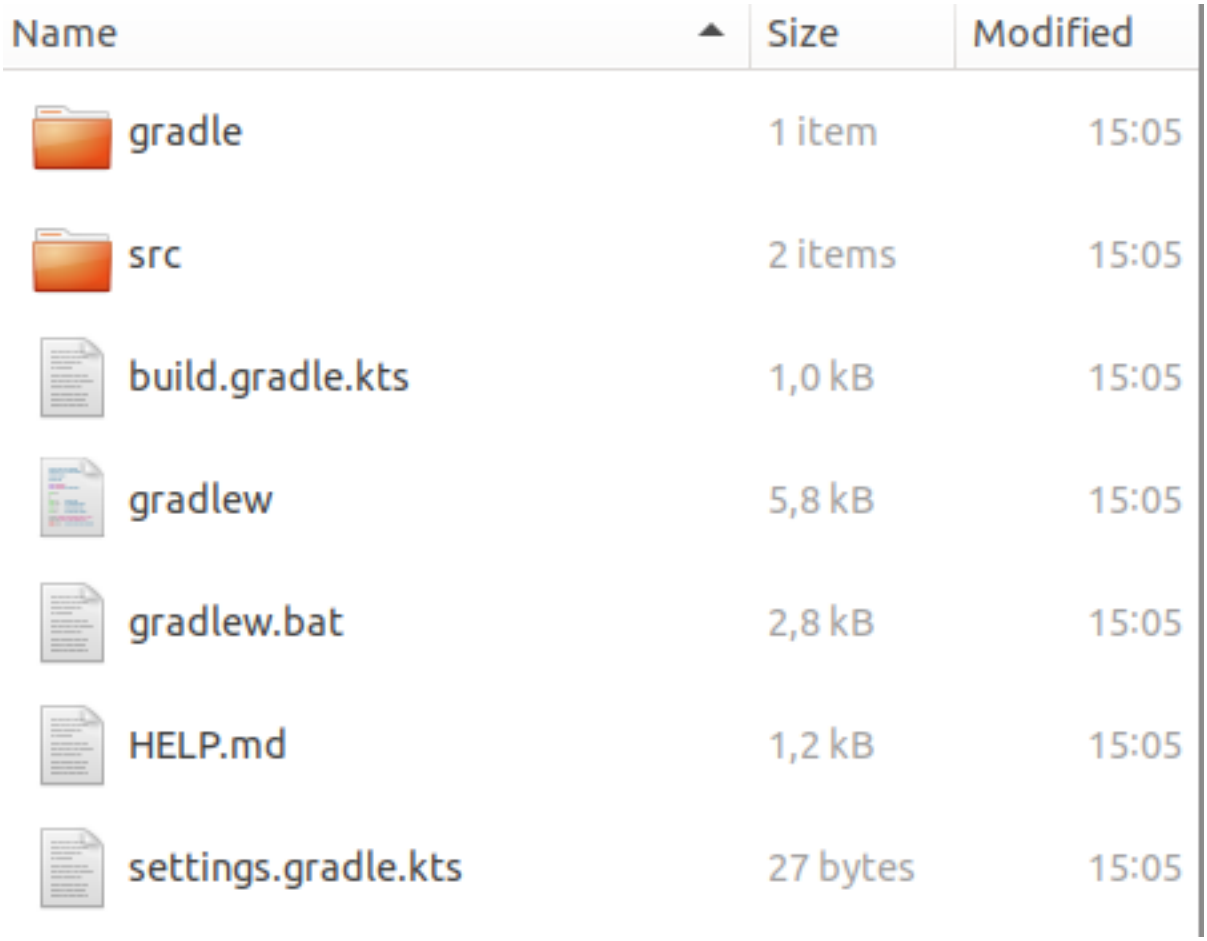


Figura 1.3: ZIP do projeto

Observe que o arquivo `HELP.md` já vem com alguns links úteis e guias a respeito de como construir serviços REST com as ferramentas escolhidas (no caso, Gradle, Spring Boot e Spring Web). Copio a seguir os links para ajudá-lo em seus estudos:

- *Building a RESTful Web Service* - https://spring.io/guides/gs/rest-service
- *Serving Web Content with Spring MVC* - https://spring.io/guides/gs/serving-web-content
- *Building REST services with Spring* - https://spring.io/guides/tutorials/bookmarks
- *Accessing Data with JPA* - https://spring.io/guides/gs/accessing-data-jpa/

Além disso, incluo mais um link na lista que pode ser útil:

- *Building web applications with Spring Boot and Kotlin* - https://spring.io/guides/tutorials/spring-boot-kotlin/

Já o projeto em si pode ser importado da forma como está em sua IDE. Ao longo deste livro, vou utilizar o IntelliJ, mas você pode usar a IDE de sua conveniência. A estrutura do projeto é a seguinte:

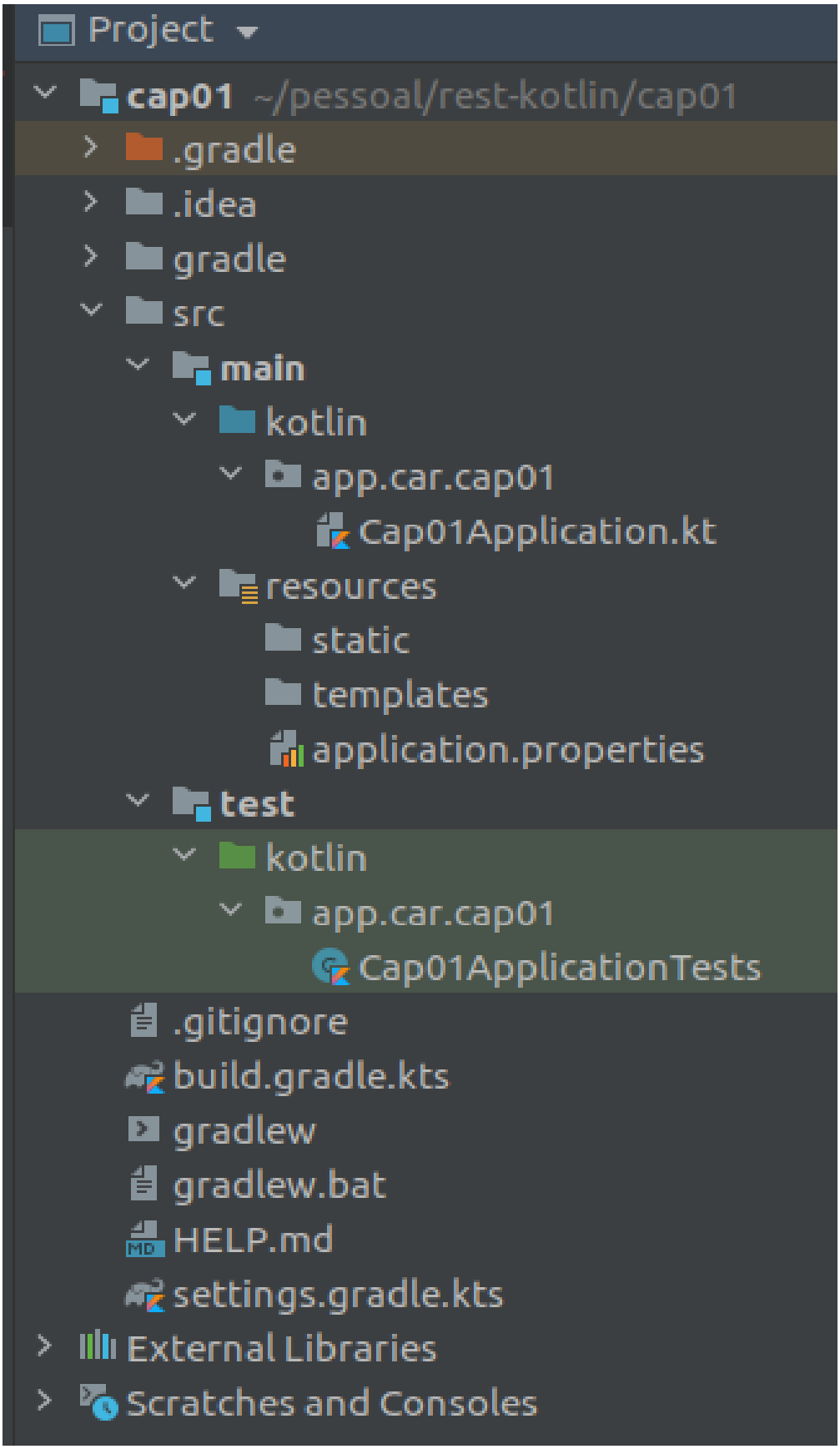

Project
cap01 ~/pessoal/rest-kotlin/cap01
.gradle
.idea
gradle
src
main
kotlin
app.car.cap01
Cap01Application.kt
resources
static
templates
application.properties
test
kotlin
app.car.cap01
Cap01ApplicationTests
.gitignore
build.gradle.kts
gradlew
gradlew.bat
HELP.md
settings.gradle.kts
External Libraries
Scratches and Consoles

Figura 1.4: Estrutura do projeto

O Spring Boot faz o download de um *web server* para nós (no momento, o padrão é o Tomcat, mas existem outros disponíveis, como Jetty e Undertow). Para inicializar o sistema, basta executar o método `main` no arquivo `Cap01Application`. O log deve ficar semelhante ao seguinte:

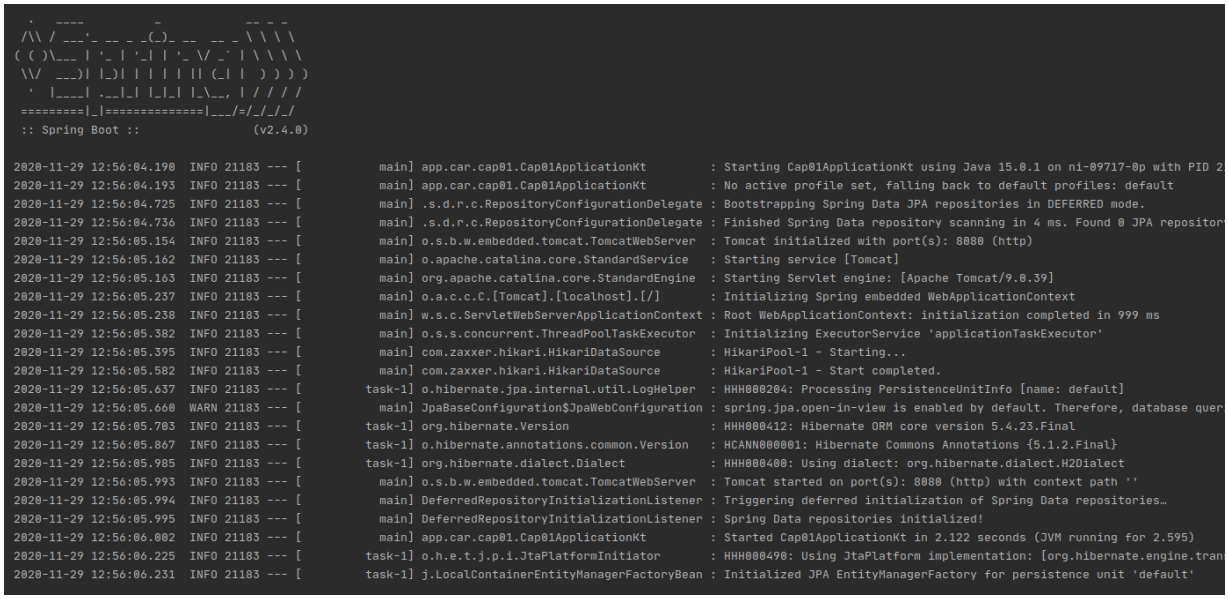

:: Spring Boot :: (v2.4.0)
2020-11-29 12:56:04.190 INFO 21183 --- [main] app.car.cap01.Cap01ApplicationKt : Starting Cap01ApplicationKt using Java 15.0.1 on ni-09717-0p with PID 2
2020-11-29 12:56:04.193 INFO 21183 --- [main] app.car.cap01.Cap01ApplicationKt : No active profile set, falling back to default profiles: default
2020-11-29 12:56:04.725 INFO 21183 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFERRED mode.
2020-11-29 12:56:04.736 INFO 21183 --- [main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 4 ms. Found 0 JPA repositor
2020-11-29 12:56:05.154 INFO 21183 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2020-11-29 12:56:05.162 INFO 21183 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2020-11-29 12:56:05.163 INFO 21183 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.39]
2020-11-29 12:56:05.237 INFO 21183 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2020-11-29 12:56:05.238 INFO 21183 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 999 ms
2020-11-29 12:56:05.382 INFO 21183 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-11-29 12:56:05.395 INFO 21183 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2020-11-29 12:56:05.582 INFO 21183 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2020-11-29 12:56:05.637 INFO 21183 --- [task-1] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name: default]
2020-11-29 12:56:05.660 WARN 21183 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database quer
2020-11-29 12:56:05.703 INFO 21183 --- [task-1] org.hibernate.Version : HHH000412: Hibernate ORM core version 5.4.23.Final
2020-11-29 12:56:05.867 INFO 21183 --- [task-1] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
2020-11-29 12:56:05.985 INFO 21183 --- [task-1] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
2020-11-29 12:56:05.993 INFO 21183 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2020-11-29 12:56:05.994 INFO 21183 --- [main] DeferredRepositoryInitializationListener : Triggering deferred initialization of Spring Data repositories…
2020-11-29 12:56:05.995 INFO 21183 --- [main] DeferredRepositoryInitializationListener : Spring Data repositories initialized!
2020-11-29 12:56:06.002 INFO 21183 --- [main] app.car.cap01.Cap01ApplicationKt : Started Cap01ApplicationKt in 2.122 seconds (JVM running for 2.595)
2020-11-29 12:56:06.225 INFO 21183 --- [task-1] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.tran
2020-11-29 12:56:06.231 INFO 21183 --- [task-1] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'

Figura 1.5: Log do projeto

1.2 O primeiro caso de uso: a listagem de novos motoristas

Vamos agora criar um primeiro serviço em REST para atender a um caso de uso. O aplicativo C.A.R. precisa de um back-end de administração, ou seja, um programa através do qual seja possível controlar quem são os motoristas, os passageiros, as corridas realizadas etc. Esse primeiro caso de uso vai precisar de um serviço para listar os motoristas cadastrados, algo que acesse o banco de dados do sistema e retorne os dados dos motoristas. Como fazer isso?

Em primeiro lugar, precisamos criar uma camada de acesso a um banco de dados e uma classe que representará os motoristas. Como prática pessoal, eu prefiro escrever os códigos em inglês (para manter a harmonia entre as palavras reservadas da linguagem e o código que estou escrevendo). Assim sendo, vou criar um pacote `domain`, um arquivo chamado `Entities.kt` e, dentro dele, uma `data class` chamada `Driver`:

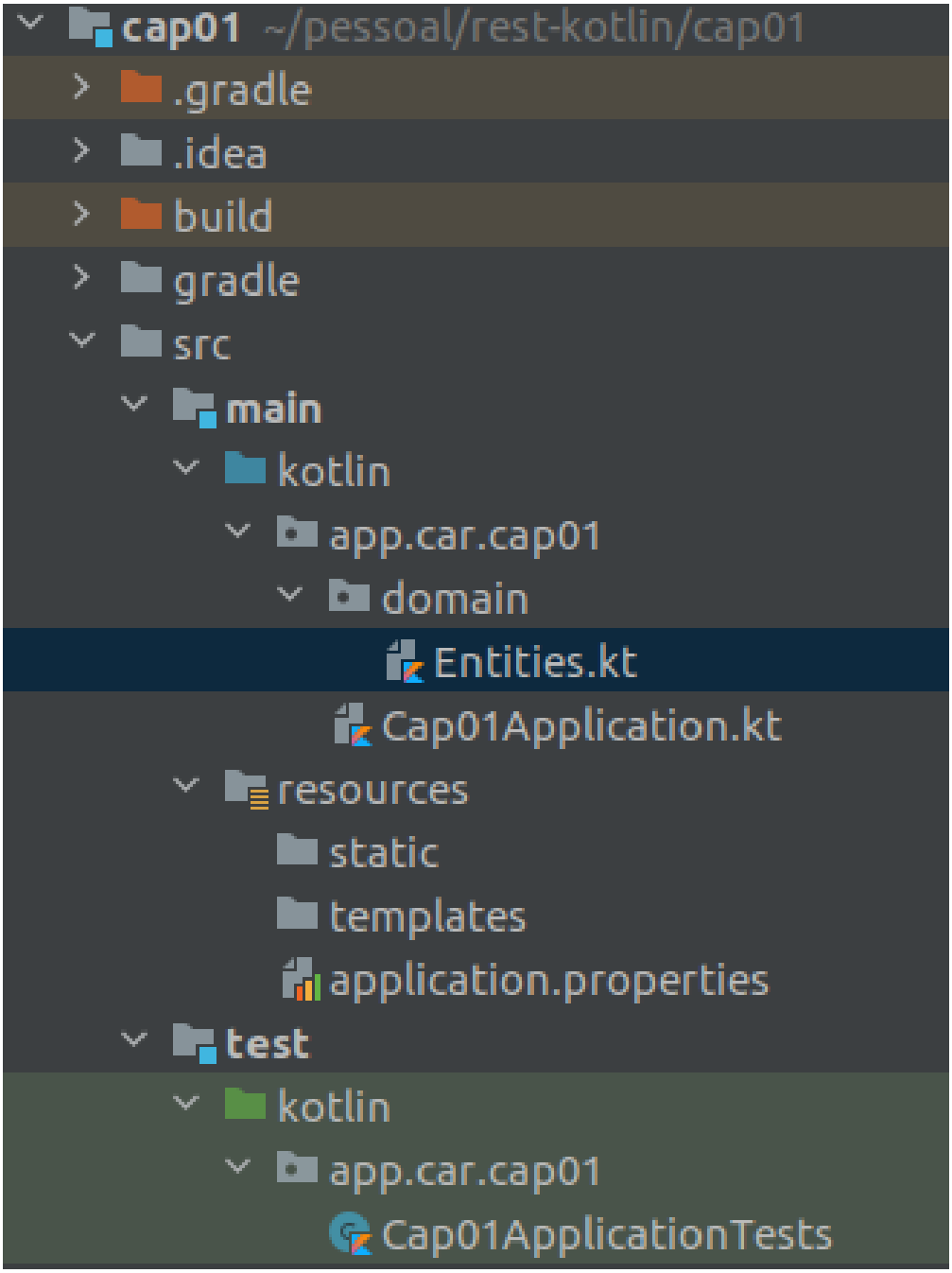


Figura 1.6: Classe que representa o motorista criada

Agora, nossa `data class` precisa de alguns campos. Vamos incluir um campo `var` para representar a identificação do motorista, que por padrão seja nulo, e também alguns campos do tipo `val` chamados `name` e `birthDate` :

```
package app.car.cap01.domain

import java.time.LocalDate

data class Driver(
        var id: Long? = null,
        val name: String,
        val birthDate: LocalDate
)
```

Precisamos modificar essa classe para refletir o estado do nosso banco de dados. Fazer isso em Kotlin também é fácil, basta anotar a classe com `javax.persistence.Entity`. Além disso, também vamos fazer com que o sistema reconheça o campo `id` como representante da chave primária da tabela correspondente. Fazemos isso com a anotação `javax.persistence.Id`:

```
package app.car.cap01.domain

import java.time.LocalDate
import javax.persistence.Entity
import javax.persistence.Id

@Entity
data class Driver(
        @Id
        var id: Long? = null,
        val name: String,
        val birthDate: LocalDate
)
```

Agora, precisamos de um meio de acesso ao banco de dados (algo semelhante aos *design patterns* DAO ou `Repository`, conforme orientado pelo *Domain-Driven Design*). Quando utilizamos o Spring Data JPA, criamos uma interface e fazemos com que ela estenda outra, a `JpaRepository`. Essa outra interface utiliza o mecanismo de *generics* do Kotlin. Na declaração dela, é utilizada a classe `Driver`, que é gerenciada pelo repositório criado e a classe da chave primária (no nosso caso, `kotlin.Long`).

Vamos criar um arquivo chamado `Repositories.kt` dentro do pacote `domain` e, dentro dele, um repositório `DriverRepository`. Realizando a extensão da interface mencionada, temos o seguinte:

```
package app.car.cap01.domain

import org.springframework.data.jpa.repository.JpaRepository

interface DriverRepository : JpaRepository<Driver, Long>
```

O próximo passo é escolher e configurar um banco de dados. Algumas vezes, em desenvolvimento, utilizamos um banco de dados em memória apenas para podermos validar algum código que desenvolvemos. Uma boa pedida para o nosso contexto é o H2, um banco de dados que inicializa junto com o nosso projeto e que já está com o *driver* incluso. Ele também conta com uma interface de administração fornecida através de uma extensão do Spring Boot, o **Spring Boot DevTools**.

Ainda precisamos incluir o DevTools no nosso projeto. Para fazer isso, incluímos a seguinte declaração de dependências no `build.gradle.kts`:

```
developmentOnly("org.springframework.boot:spring-boot-devtools")
```

Da forma como está, já é possível verificar se está tudo certo. O Spring Boot, ao detectar esses componentes, configura o H2 para inicializar em memória com um banco nomeado aleatoriamente, com um usuário `sa` e sem senha. O Spring Boot DevTools disponibiliza uma interface de gerenciamento do H2 chamado H2 console. Inicialize o projeto e cheque o console para verificar qual é a URL de conexão do banco:

```
: HikariPool-1 - Start completed.
: H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem:ad373c01-e3dd-4d1f-8891-37c3479096df'
: Initializing ExecutorService 'applicationTaskExecutor'
: HHH000204: Processing PersistenceUnitInfo [name: default]
```

Figura 1.7: URL de conexão do banco H2

Copie a URL e abra no seu *browser* a URL http://localhost:8080/h2-console/ para ver a seguinte tela:

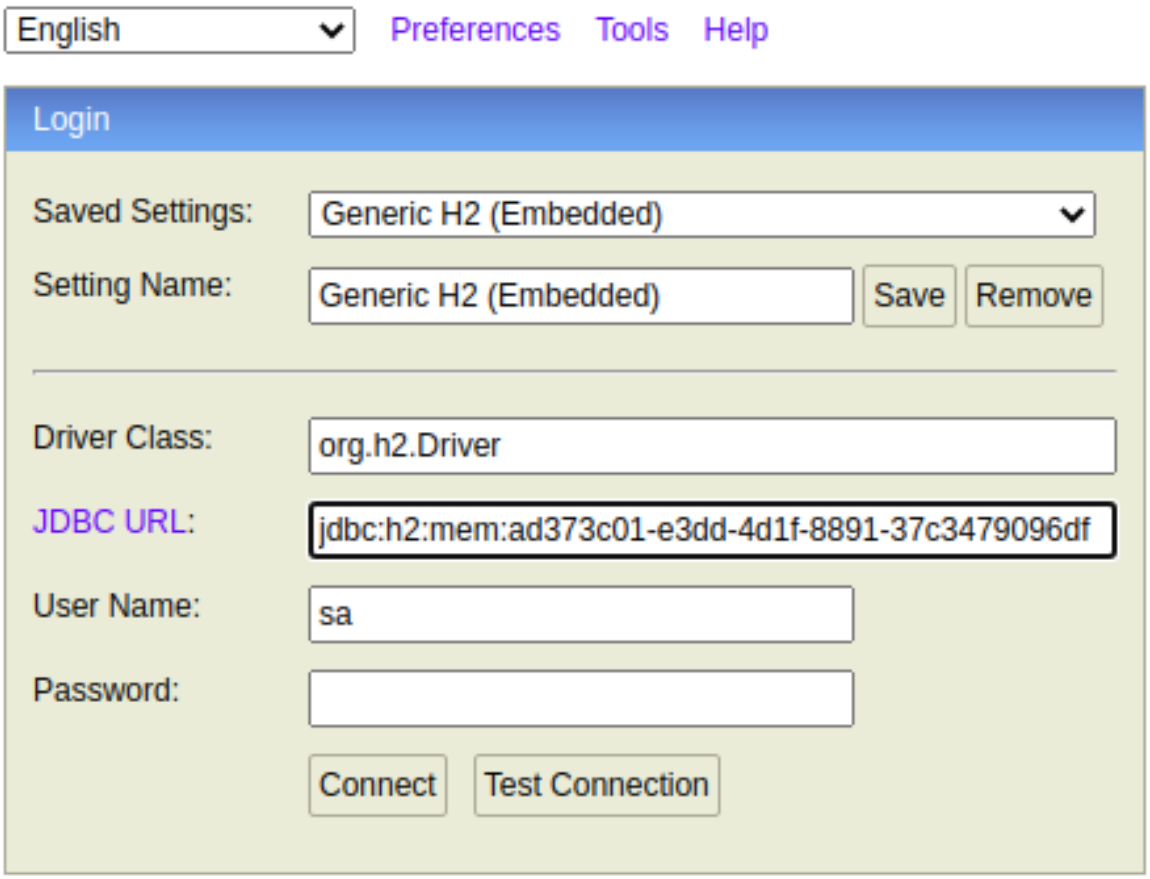


Figura 1.8: Tela de login do H2 Console

Certifique-se de que as configurações estão assim:

- Driver Class: org.h2.Driver
- JDBC URL: (a que você copiou do console)
- User Name: sa
- Password: (em branco)

Ao clicar em *Connect*, você deve ver uma tela semelhante à seguinte:

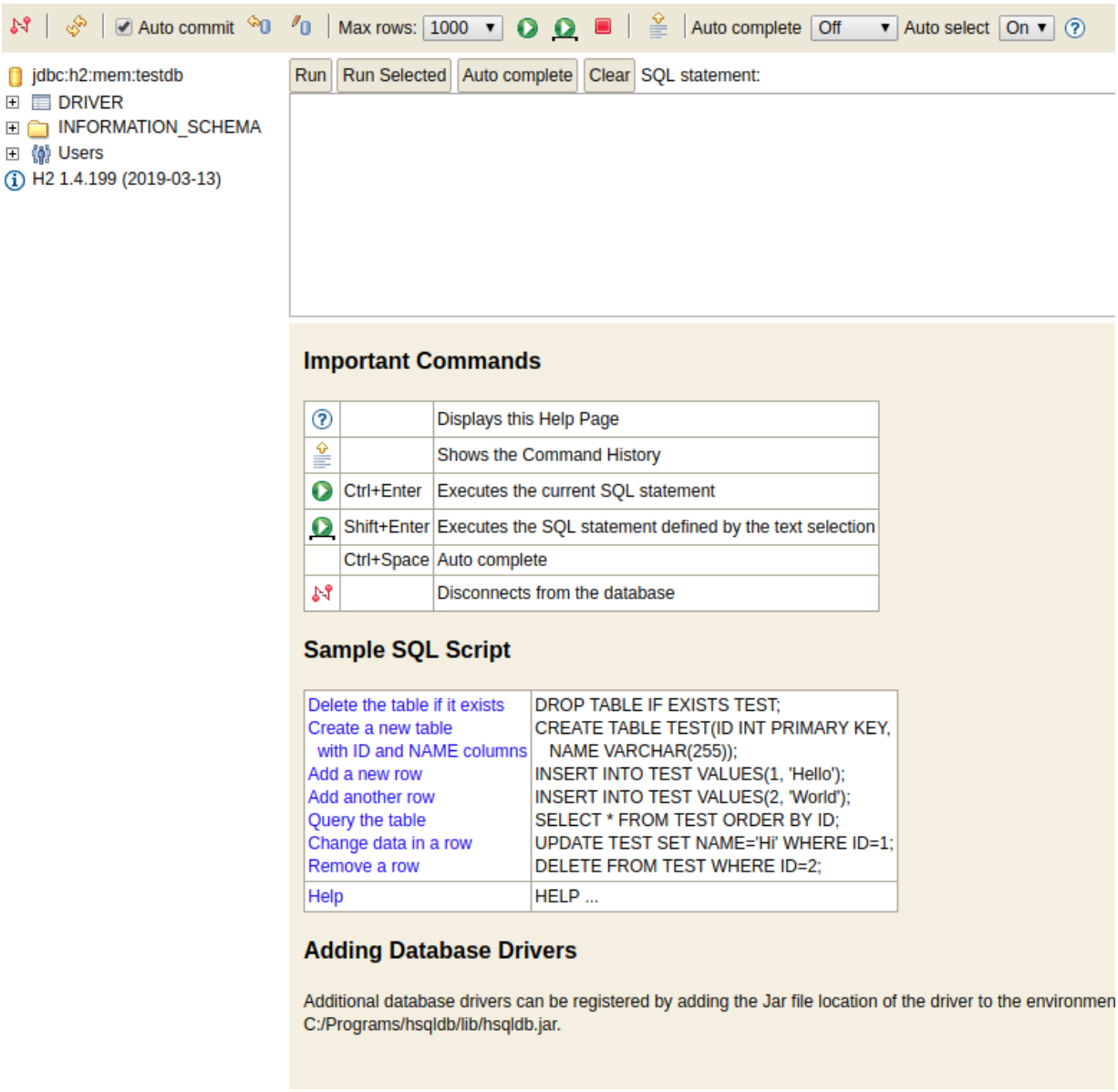


Figura 1.9: H2 Console

Observe que uma tabela chamada `Driver` já está criada. Se você clicar no botão `+`, ao lado do nome da tabela, verá que os campos mapeados anteriormente também já estão criados:

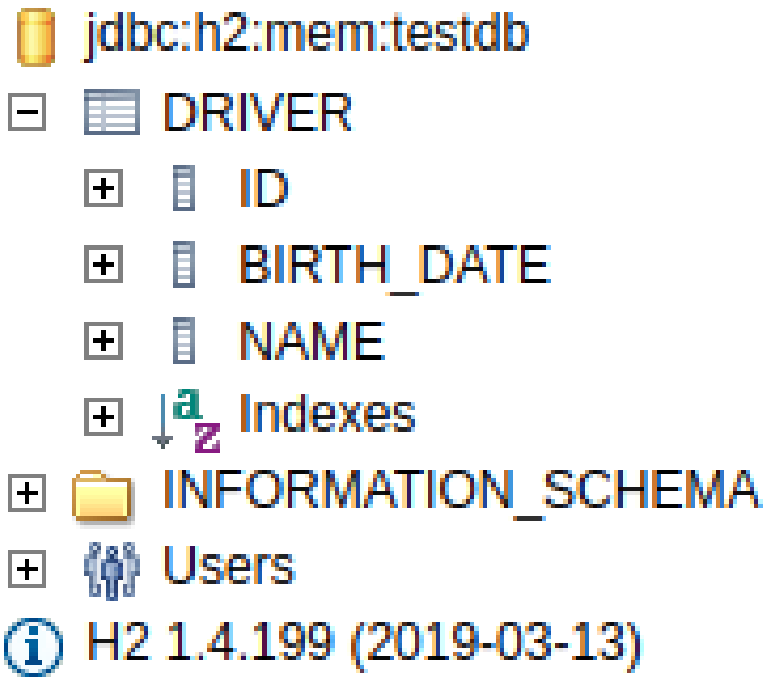


Figura 1.10: Tabela Driver criada

Verificamos, assim, que nossa infraestrutura já está pronta. Ótimo! Só resta mesmo o serviço REST.

Para criar o serviço, vamos criar um novo pacote no projeto chamado `interfaces` (observe que o nome está no plural para não conflitar com a palavra `interface`, que é reservada no Kotlin). Ele vai ficar no mesmo nível do pacote `domain`. Dentro dele, vamos criar uma classe chamada `DriverAPI`:

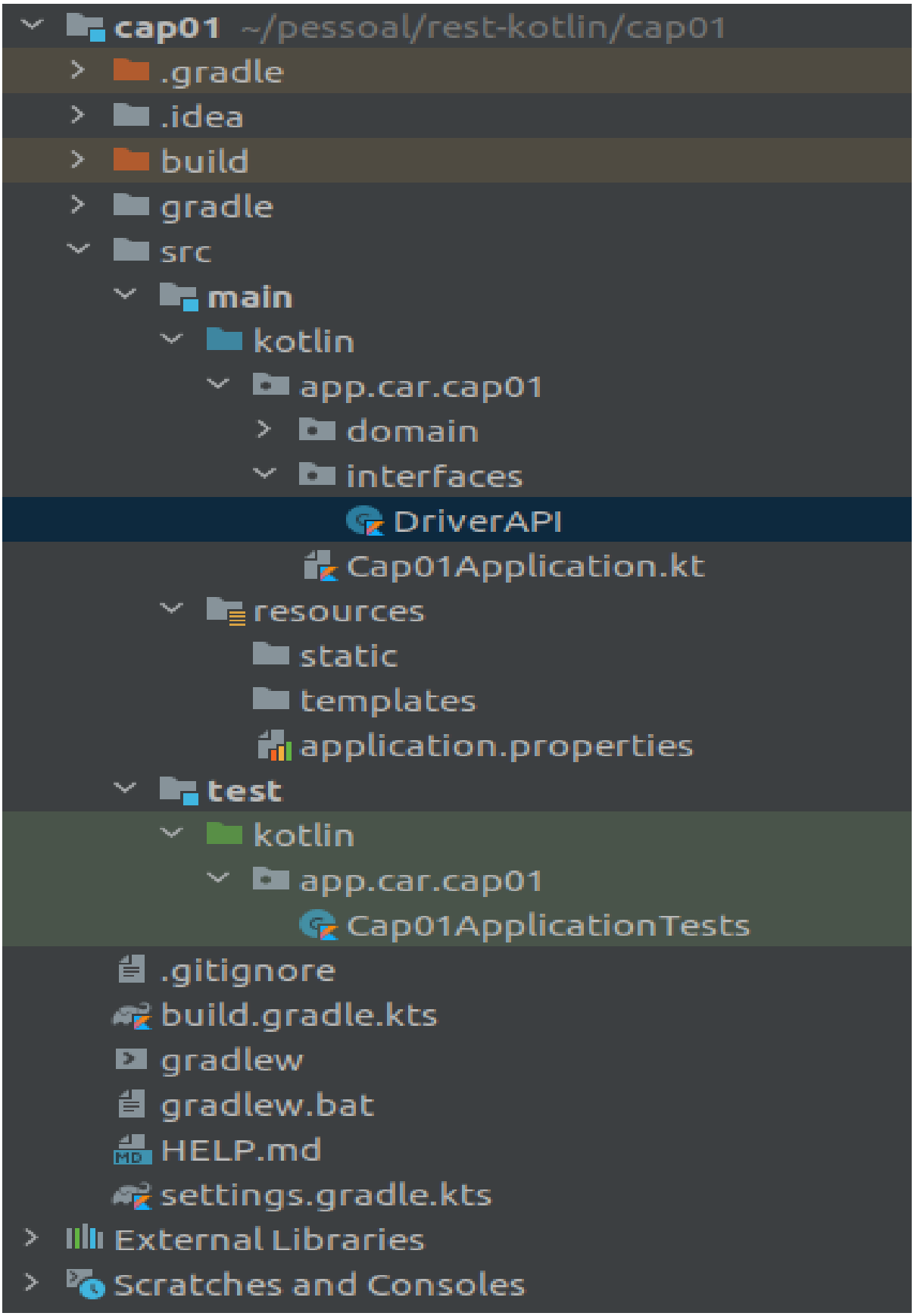

cap01 ~/pessoal/rest-kotlin/cap01
.gradle
.idea
build
gradle
src
main
kotlin
app.car.cap01
domain
interfaces
DriverAPI
Cap01Application.kt
resources
static
templates
application.properties
test
kotlin
app.car.cap01
Cap01ApplicationTests
.gitignore
build.gradle.kts
gradlew
gradlew.bat
HELP.md
settings.gradle.kts
External Libraries
Scratches and Consoles

Figura 1.11: Classe DriverAPI criada

Dentro dessa classe, vamos criar um método que nos devolva todos os motoristas cadastrados no repositório. Esse método vai se chamar `listDrivers`, vai retornar uma `List` com objetos do tipo `Driver` e não terá parâmetro algum:

```
package app.car.cap01.interfaces

import app.car.cap01.domain.Driver

class DriverAPI {

    fun listDrivers() : List<Driver> = emptyList()

}
```

Para que essa classe seja reconhecida pelo Spring como um serviço REST, é necessário que ela esteja anotada com `@Service` e `@RestController`:

```
import org.springframework.stereotype.Service
import org.springframework.web.bind.annotation.RestController

@Service
@RestController
class DriverAPI {
// Restante do código omitido
```

Agora, precisamos de alguma forma expor essa informação para o mundo. Para fazer isso, precisamos utilizar um método HTTP qualquer. É aqui que começam a entrar os conceitos de REST, mas quais são os métodos e como escolher?

1.3 Quais são os métodos HTTP e como escolher entre eles?

O protocolo HTTP disponibiliza, até o momento, nove métodos HTTP (sendo que sete deles podem ser utilizados em REST). Cada um tem a função de nos fornecer uma forma de interação com o servidor de acordo com nossa intenção ou necessidade; e destes sete, são cinco os principais, que podem ser relacionados com operações semelhantes à interação com um banco de dados. São os seguintes:

- `GET` - Para recuperação de dados.
- `POST` - Para criação de dados.
- `PUT` e `PATCH` - Para atualização de dados.
- `DELETE` - Para apagar dados.

Observe, no entanto, que essas definições são aproximações **muito** simplistas que utilizo aqui apenas para fins didáticos. Vou entrar em mais detalhes a respeito dos métodos HTTP e sua utilização no capítulo 2.

Como queremos **listar** os dados dos motoristas, vamos utilizar o método `GET`. É possível fazer isso em Spring com a anotação `GetMapping`:

```
// Restante dos imports omitidos
import org.springframework.web.bind.annotation.GetMapping

@Service
@RestController
class DriverAPI {

    @GetMapping
    fun listDrivers() : List<Driver> = emptyList()

}
```

O próximo passo é determinar qual será o tipo de mídia retornado por essa API. Existem vários tipos de mídia disponíveis, mas geralmente essa é uma decisão fácil de ser tomada, pois cada uma delas tem uma característica específica. No nosso caso, desejamos servir os dados dos motoristas em questão, ou seja, precisamos

servir informações estruturadas, de forma que a API se comporte como se fosse um catálogo. Temos duas linguagens principais para fazer isso: XML (`eXtensible Markup Language` , ou linguagem de marcação extensível) e JSON (`JavaScript Object Notation` , ou notação de objetos JavaScript). Acontece que, entre elas, JSON foi a que se tornou a língua franca para servir dados estruturados em serviços REST, o que facilita muito a escolha. Isso se deve em grande parte ao tamanho de *payloads* equivalentes, que é maior em arquivos XML que usam muitas *tags*, o que onera redes mais limitadas, como as redes móveis.

Para informar o sistema do Spring de que essa API deverá produzir dados em JSON, utilizamos a anotação `@RequestMapping` , junto do parâmetro `produces` . Esse parâmetro recebe uma lista de Strings como dados, mas as strings aceitas estão populadas como constantes na classe `MediaType` do Spring. Quando desejamos utilizar JSON, o valor a ser empregado é `APPLICATION_JSON_VALUE` , de forma que a classe fica assim:

```
import app.car.cap01.domain.Driver
import org.springframework.http.MediaType
import org.springframework.stereotype.Service
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@Service
@RestController
@RequestMapping(produces = [MediaType.APPLICATION_JSON_VALUE])
class DriverAPI {

    @GetMapping
    fun listDrivers() : List<Driver> = emptyList()

}
```

Por último, precisamos informar ao Spring qual URL vamos utilizar para mapear esse recurso. Esse é um passo **muito** importante, pois em REST cada recurso tem a sua própria URL e esta deve ser

significativa, de forma a estimular a usabilidade desses serviços. Uma vez que estamos listando instâncias da classe `Driver`, vamos adotar a URL `/drivers`.

> **USAR AS URLS NO SINGULAR OU PLURAL?**
>
> Este é um assunto sobre o qual não existe definição formal (ou mesmo informal). Você pode definir suas URLs no singular (no nosso exemplo, seria `/driver`) ou no plural (`/drivers`). O único consenso existente a esse respeito na comunidade é que deve haver um padrão para todas as APIs de um mesmo sistema, ou seja, todas no singular ou todas no plural.

Para mapear essa listagem usando a URL, basta colocá-la como String na anotação `@GetMapping`:

```
@GetMapping("/drivers")
fun listDrivers() : List<Driver> = emptyList()
```

Agora, basta inicializar o sistema e navegar para a URL `http://localhost:8080/drivers`. Você deve ver em seu browser o seguinte:

```
[]
```

Essa é uma lista vazia em JSON. Note que ela é a representação do que está no método `listDrivers`, ou seja, no retorno dele, temos mesmo uma lista vazia. O que resta agora é fazer com que a classe `DriverAPI` tenha acesso à interface `DriverRepository`. O Spring é especializado nisso - como a `DriverAPI` está anotada com `@Service`, basta fazer a inclusão da `DriverRepository` como parâmetro do construtor da `DriverAPI`:

```
package app.car.cap01.interfaces

import app.car.cap01.domain.Driver
import app.car.cap01.domain.DriverRepository
import org.springframework.http.MediaType
```

```
import org.springframework.stereotype.Service
import org.springframework.web.bind.annotation.GetMapping
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RestController

@Service
@RestController
@RequestMapping(produces = [MediaType.APPLICATION_JSON_VALUE])
class DriverAPI(
        val driverRepository: DriverRepository
) {

    @GetMapping("/drivers")
    fun listDrivers() : List<Driver> = emptyList()

}
```

A anotação @Autowired e o Kotlin

Se você já estiver acostumado com o uso de Spring Boot na linguagem Java, pode ter notado a ausência da anotação `@Autowired` no construtor da DriverAPI. Quando utilizamos Spring Boot com Kotlin, as dependências são reconhecidas automaticamente, dispensando o uso da anotação.

Precisamos então utilizar o `DriverRepository` para retornar os dados do banco de dados, o que é feito através do método `findAll`. O método modificado fica assim:

```
@GetMapping("/drivers")
fun listDrivers() = driverRepository.findAll()
}
```

Ao reinicializar o projeto, o resultado produzido pela URL `/drivers` continua igual. Mas agora, ele está vinculado ao conteúdo do banco de dados. Vamos acessar o H2 Console para popular alguns dados no sistema. Ao entrar, digite o seguinte:

```
INSERT INTO DRIVER(ID, BIRTH_DATE, NAME)
VALUES (1, '1986-08-18', 'Alexandre Saudate')
```

Ao clicar em `Run`, a parte de baixo do H2 Console muda e fica com o seguinte texto:

```
INSERT INTO DRIVER(ID, BIRTH_DATE, NAME)
VALUES (1, '1986-08-18', 'Alexandre Saudate');
Update count: 1
(1 ms)
```

A linha `Update count: 1` diz que uma linha foi atualizada (nesse caso, criada). Agora, ao voltar para o browser e atualizar a página, você deve visualizar algo semelhante ao que se vê na imagem:

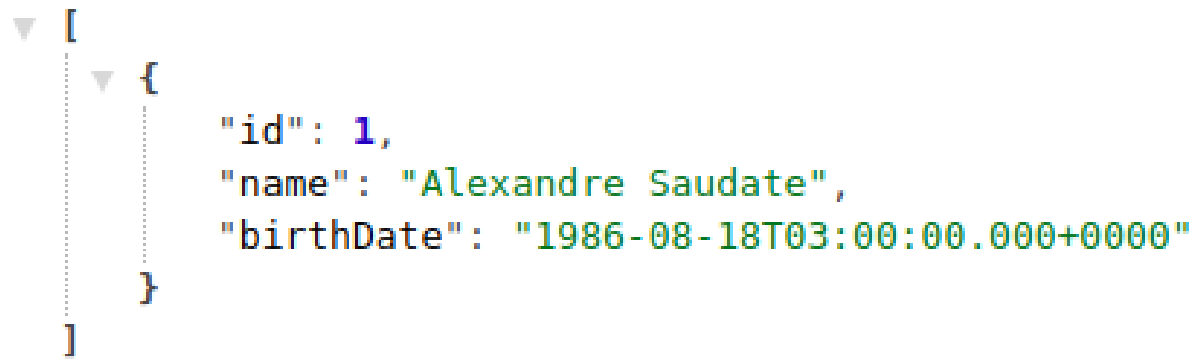


Figura 1.12: Dados sendo retornados da API

Vamos inserir mais um registro para verificar o retorno em lista. Para isso, utilize o seguinte:

```
INSERT INTO DRIVER(ID, BIRTH_DATE, NAME)
VALUES (2, '1981-06-03', 'McLOVIN')
```

Ao atualizar o browser, visualizamos algo assim:

```
[
   {
      "id": 1,
      "name": "Alexandre Saudate",
      "birthDate": "1986-08-18T03:00:00.000+0000"
```

```
   },
   {
      "id": 2,
      "name": "McLOVIN",
      "birthDate": "1981-06-03T03:00:00.000+0000"
   }
]
```

Conclusão

Neste capítulo, você viu como criar um projeto com Spring Boot (utilizando Spring Initializr), como utilizar as ferramentas do Spring Boot para criar um banco de dados em memória, popular esse banco e servir os dados em uma API REST, mas ainda há muito o que ser feito. Nos próximos capítulos, veremos como criar esses dados através de uma API REST e quais métodos podemos utilizar para esse propósito. Vamos em frente?

Capítulo 2
Expandindo o nosso serviço inicial

"Sorte é o que acontece quando a preparação encontra a oportunidade." - Sêneca.

Nosso primeiro serviço está funcionando, já é interoperável, mas não é muito funcional, pois precisamos, de alguma forma, permitir a recuperação dos dados de apenas um motorista (afinal de contas, queremos ter *muitos*, certo?), o cadastro e a atualização dos dados. Temos muito por onde seguir.

2.1 Recuperando os dados de um motorista específico

Vamos revisitar por um momento a URL utilizada para a listagem dos motoristas. A URL foi `/drivers`, ou seja, uma palavra no plural para indicar uma coleção de dados. Em REST, o nome que se dá às informações trafegadas é **recurso**. Quando definimos a URL e o tipo de mídia (JSON), nós criamos uma definição de um recurso.

Acontece que as definições de recursos são hierárquicas. Espera-se que um motorista em particular esteja contido nessa lista, e essa lista, por si só, é um recurso. Se quisermos criar um serviço para recuperar os dados de um motorista em particular, devemos criar um novo recurso - desta vez, hierarquicamente dependente de `/drivers`. Se para criar recursos utilizamos URLs e um motorista diferente é um recurso diferente (porém subordinado ao recurso "pai" que é a listagem de motoristas), então precisamos estender a URL `/drivers` com algo que possa identificá-lo de maneira única. Vamos revisitar a listagem de motoristas que foi retornada anteriormente:

```
[
   {
      "id": 1,
      "name": "Alexandre Saudate",
      "birthDate": "1986-08-18T03:00:00.000+0000"
   },
   {
      "id": 2,
      "name": "McLOVIN",
      "birthDate": "1981-06-03T03:00:00.000+0000"
   }
]
```


Observe que o campo `id` pode ser utilizado para esse fim! Mas ainda existe um problema: o campo `id` é dinâmico, novos IDs são criados conforme os dados são populados no servidor. Logo, não existe um número determinado de IDs, eles serão modificados o tempo todo.

Para resolver esse problema, em REST, temos o conceito de **path parameters**: parâmetros que são colocados na própria URL. Dessa forma, podemos dizer que a URL do nosso novo recurso seria algo como `/drivers/{id}`, onde o valor que está entre chaves é nossa variável.

Para expressar essa mudança, vamos criar um novo método na nossa classe `DriverAPI`, que localiza o motorista específico:

```
@GetMapping("/drivers/{id}")
public Driver findDriver(){
```

Agora, precisamos vincular algo ao `id`. Como vamos utilizar o `ID` do banco de dados para também representar o `id` dos nossos recursos, temos que utilizar o mesmo tipo, `Long`. Por último, temos que vincular o texto `{id}` ao parâmetro, o que é feito utilizando a anotação `@PathVariable`:

```
import org.springframework.web.bind.annotation.PathVariable;
// Restante do código omitido...
```

```
@GetMapping("/drivers/{id}")
fun findDriver(@PathVariable("id") id: Long) = //...
```

Em seguida, só resta recuperar o motorista específico utilizando o método `findById` do repositório. Note, no entanto, que esse método retorna um `java.util.Optional`, e não um `Driver`. Para recuperarmos o `Driver` presente no `Optional`, utilizamos o método `get()`:

```
@GetMapping("/drivers/{id}")
fun findDriver(@PathVariable("id") id: Long) =
    driverRepository.findById(id).get()
```

Para testar o resultado, precisamos inicializar novamente nosso programa, entrar no H2 Console e reinserir os dados. Utilize o seguinte SQL:

```
INSERT INTO DRIVER(ID, BIRTH_DATE, NAME)
   VALUES (1, '1986-08-18', 'Alexandre Saudate');
INSERT INTO DRIVER(ID, BIRTH_DATE, NAME)
   VALUES (2, '1981-06-03', 'McLOVIN');
```

Feito isso, navegue para as URLs correspondentes para verificar o resultado; primeiro, para `http://localhost:8080/drivers/1`:

```
{
   "id": 1,
   "name": "Alexandre Saudate",
   "birthDate": "1986-08-18T03:00:00.000+0000"
}
```

E depois para `http://localhost:8080/drivers/2`:

```
{
   "id": 2,
   "name": "McLOVIN",
   "birthDate": "1981-06-03T03:00:00.000+0000"
}
```

2.2 Conhecendo os códigos de status

O serviço está funcionando para os casos básicos, ou seja, para recursos que existem no sistema, os motoristas 1 e 2. Mas e quanto a motoristas que não existem? Se você navegar até a URL `http://localhost:8080/drivers/3` , por exemplo, verá o seguinte:

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Dec 02 08:03:18 BRT 2020
There was an unexpected error (type=Internal Server Error, status=500).
No value present
java.util.NoSuchElementException: No value present
at java.base/java.util.Optional.get(Optional.java:148)
at app.car.cap02.interfaces.DriverAPI.findDriver(DriverAPI.kt:23)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.base/java.lang.reflect.Method.invoke(Method.java:566)
at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:197)
at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:141)
at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:106)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:893)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:807)
at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87)
at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1061)
at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:961)
at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:1006)
at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:898)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:626)
at org.springframework.web.servlet.FrameworkServlet.service(FrameworkServlet.java:883)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:733)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
at org.springframework.web.filter.RequestContextFilter.doFilterInternal(RequestContextFilter.java:100)
at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:119)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
at org.springframework.web.filter.FormContentFilter.doFilterInternal(FormContentFilter.java:93)
at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:119)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
at org.springframework.web.filter.CharacterEncodingFilter.doFilterInternal(CharacterEncodingFilter.java:201)
at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:119)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:193)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166)
at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:202)
at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:97)
at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:542)
at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:143)

Figura 2.1: Dados sendo retornados da API

Observe especificamente a linha:

```
There was an unexpected error (type=Internal Server Error, status=500).
```

O protocolo HTTP define alguns tipos de códigos de status, cada um condicionado a uma hierarquia diferente. As hierarquias são as seguintes:

- Os códigos que começam com 1 (1xx) são informacionais - Mostram ao cliente que a requisição foi recebida e que algo está sendo executado. Não são muito utilizados.
- Os que começam com 2 (2xx) são códigos de sucesso - Mostram ao cliente que a requisição finalizou seu processamento com sucesso. O código 200 é uma resposta genérica de sucesso, e os outros propõem ao cliente que este faça tratamentos mais complexos, no entanto, sempre indicando que o processamento foi feito com sucesso.
- Os que começam com 3 (3xx) são códigos de redirecionamento - Mostram ao cliente que o resultado final da requisição depende de outras ações.
- Os que começam com 4 (4xx) são códigos de falha - Mostram falhas originadas por algo de errado que o cliente fez, como erros de validação, por exemplo.
- Os que começam com 5 (5xx) são códigos de falha - Mostram falhas originadas por uma condição não tratada pelo servidor.

Conforme a narrativa deste livro for evoluindo, veremos os códigos necessários para realizarmos os tratamentos. Por ora, vamos nos ater ao problema que temos em mãos: nosso serviço retornou o código 500 para o caso de não haver o motorista 3. Observe que a descrição desse status é `Internal Server Error`, ou *Erro interno do servidor*, o que indica que o servidor não tratou corretamente a execução do método `get()` no caso de o `Optional` não conter valor algum armazenado dentro dele.

Observe o contexto como um todo: ao passar uma requisição para um recurso inexistente, o cliente provocou um defeito no servidor. Isso deveria ser tratado como uma falha originada pelo cliente. Por isso, vamos utilizar uma anotação mais condizente com esse caso - a `404`, que tem o descritivo `Not Found`, ou *Não Encontrado*. Este código de status foi criado justamente para esse cenário, então não há código mais apropriado do que este.

Para adaptar o código, vamos usar uma facilidade da classe `Optional`, o método `orElseThrow()`. Ele recebe como parâmetro uma

função lambda que retorna uma exceção, que é a que vamos lançar. O Spring fornece uma exceção especial para casos como o nosso, a `ResponseStatusException`. Ela recebe como parâmetro um dos valores presentes no enum `HttpStatus`, que contém uma lista dos códigos de status aceitos oficialmente pelo protocolo HTTP. Assim sendo, podemos modificar nosso código para ficar da seguinte forma:

```
import org.springframework.http.HttpStatus;
import org.springframework.web.server.ResponseStatusException;
// ...

@GetMapping("/drivers/{id}")
fun findDriver(@PathVariable("id") id: Long) =
    driverRepository.findById(id)
        .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }
```

Assim, ao realizar o teste de inicializar nosso projeto e acessar novamente a URL `http://localhost:8080/drivers/3`, vemos o seguinte:

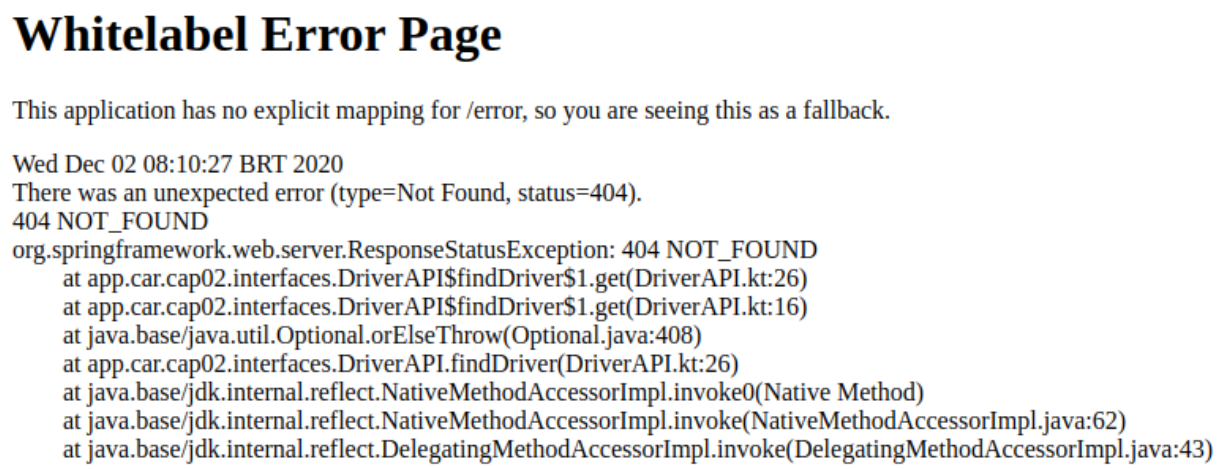

Whitelabel Error Page
This application has no explicit mapping for /error, so you are seeing this as a fallback.
Wed Dec 02 08:10:27 BRT 2020
There was an unexpected error (type=Not Found, status=404).
404 NOT_FOUND
org.springframework.web.server.ResponseStatusException: 404 NOT_FOUND
at app.car.cap02.interfaces.DriverAPI$findDriver$1.get(DriverAPI.kt:26)
at app.car.cap02.interfaces.DriverAPI$findDriver$1.get(DriverAPI.kt:16)
at java.base/java.util.Optional.orElseThrow(Optional.java:408)
at app.car.cap02.interfaces.DriverAPI.findDriver(DriverAPI.kt:26)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

Figura 2.2: Código 404 sendo retornado da API

Como você pode observar, essa página de erro agora contém no miolo o seguinte status:

```
There was an unexpected error (type=Not Found, status=404).
404 NOT_FOUND
```

2.3 Utilizando um cliente adequado - Introdução ao Postman

Para utilizar APIs, precisamos utilizar um cliente que seja adequado para controlarmos a passagem de informações para o servidor e ver o que está sendo retornado - afinal, precisamos também controlar os dados que serão fornecidos para os programas que vamos escrever e que vão consumir APIs REST. Gostaria de introduzir aqui uma das ferramentas mais difundidas e utilizadas para lidar com essas APIs, o Postman, que está disponível em https://www.getpostman.com/. Uma vez instalado e executado, você deve ver uma tela semelhante à seguinte:

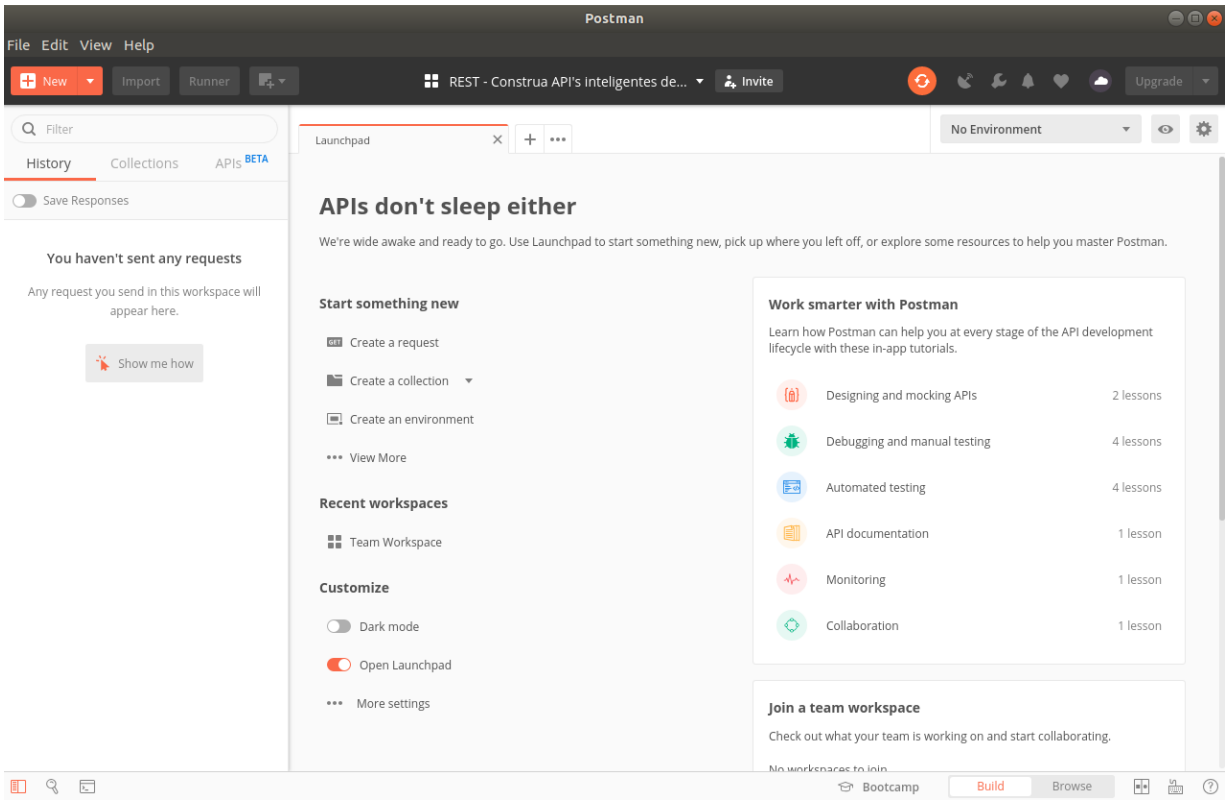


Figura 2.3: Tela inicial do Postman

Vamos agora utilizar o Postman para criar as mesmas requisições que fizemos via *browser*. Para isso, clique no botão `New`, que fica no canto superior à esquerda:

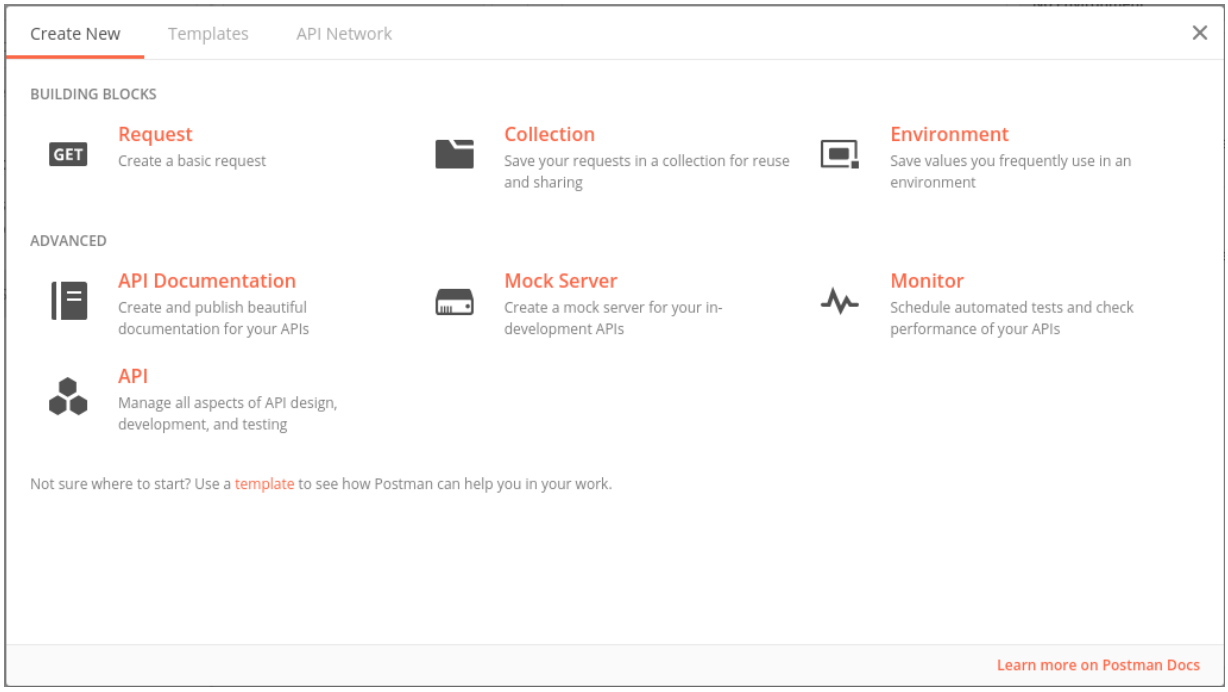

Create New
Templates
API Network
BUILDING BLOCKS
GET
Request
Create a basic request
Collection
Save your requests in a collection for reuse and sharing
Environment
Save values you frequently use in an environment
ADVANCED
API Documentation
Create and publish beautiful documentation for your APIs
Mock Server
Create a mock server for your in-development APIs
Monitor
Schedule automated tests and check performance of your APIs
API
Manage all aspects of API design, development, and testing
Not sure where to start? Use a template to see how Postman can help you in your work.
Learn more on Postman Docs

Figura 2.4: Criando request inicial no Postman

Clique no botão `Request` . Ao fazer isso, você deve observar o seguinte:

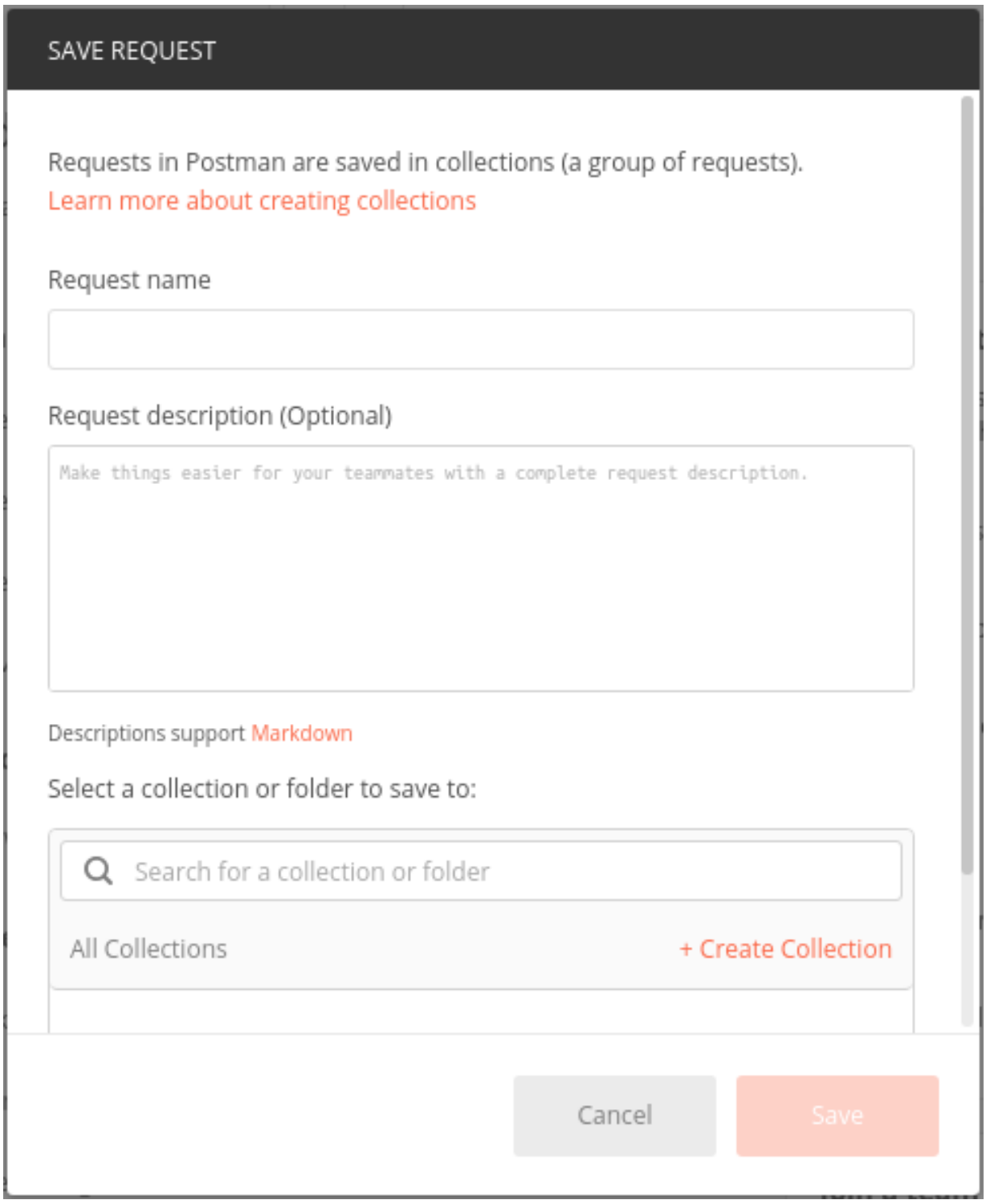

SAVE REQUEST
Requests in Postman are saved in collections (a group of requests).
Learn more about creating collections
Request name
Request description (Optional)
Make things easier for your teammates with a complete request description.
Descriptions support Markdown
Select a collection or folder to save to:
Search for a collection or folder
All Collections
+ Create Collection
Cancel
Save

Figura 2.5: Salvando a request inicial no Postman

Nessa tela, preencha o campo `Request Name` com o valor `listar motoristas`. No campo `Select a collection or folder to save to`, digite

CAR e depois clique no botão `Create Collection "CAR"` . Na sequência, clique no botão `Save to CAR` . A tela deve ficar assim:

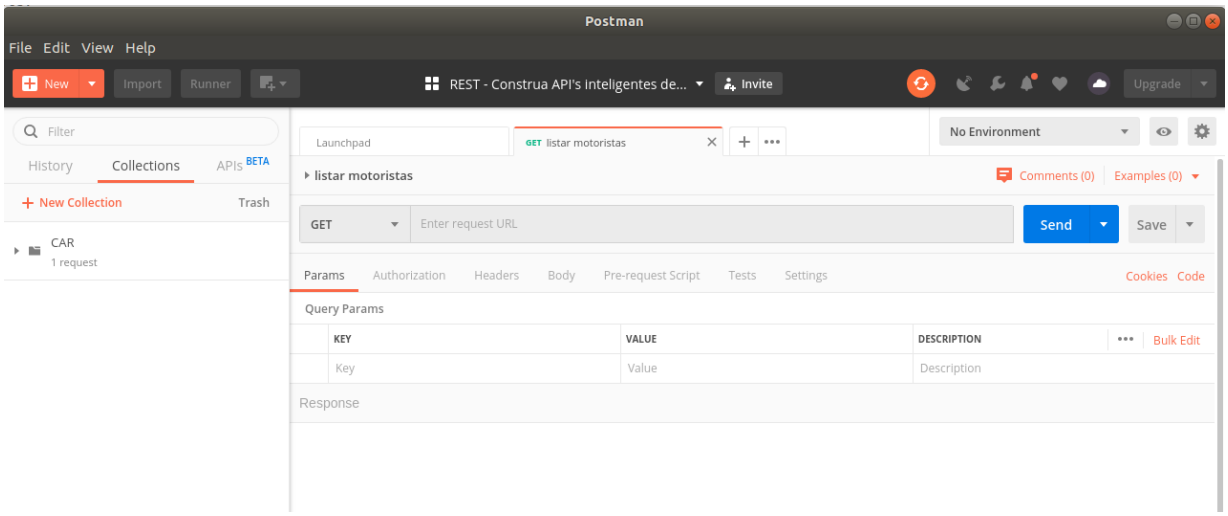


Figura 2.6: Salvando a request inicial no Postman

Finalmente, no campo onde está escrito `Enter request URL` , coloque a URL da API para listar todos os motoristas, `http://localhost:8080/drivers` . Feito isto, clique em `Send` . Você deve ver uma tela semelhante à seguinte:

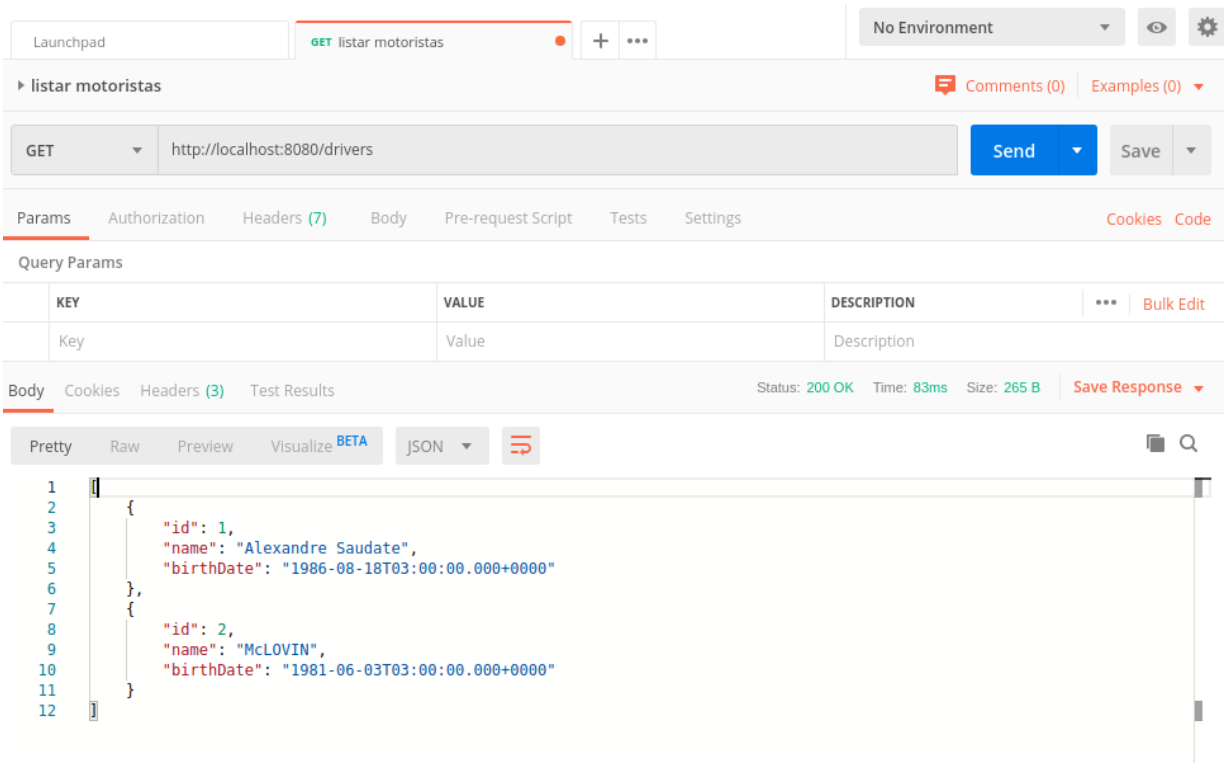


Figura 2.7: Primeira listagem com Postman

Como você pode observar, o conteúdo continua aparecendo. Dessa vez, é possível ler um campo chamado `Status` com o valor `200 OK` no centro e à direita. Esse é o código de status da resposta dada pela API, uma resposta genérica de sucesso.

Vamos agora testar pelo Postman um retorno com o código `404`. Altere a URL para `http://localhost:8080/drivers/3` e aperte `Send`. Uma resposta como a seguinte pode ser visualizada:

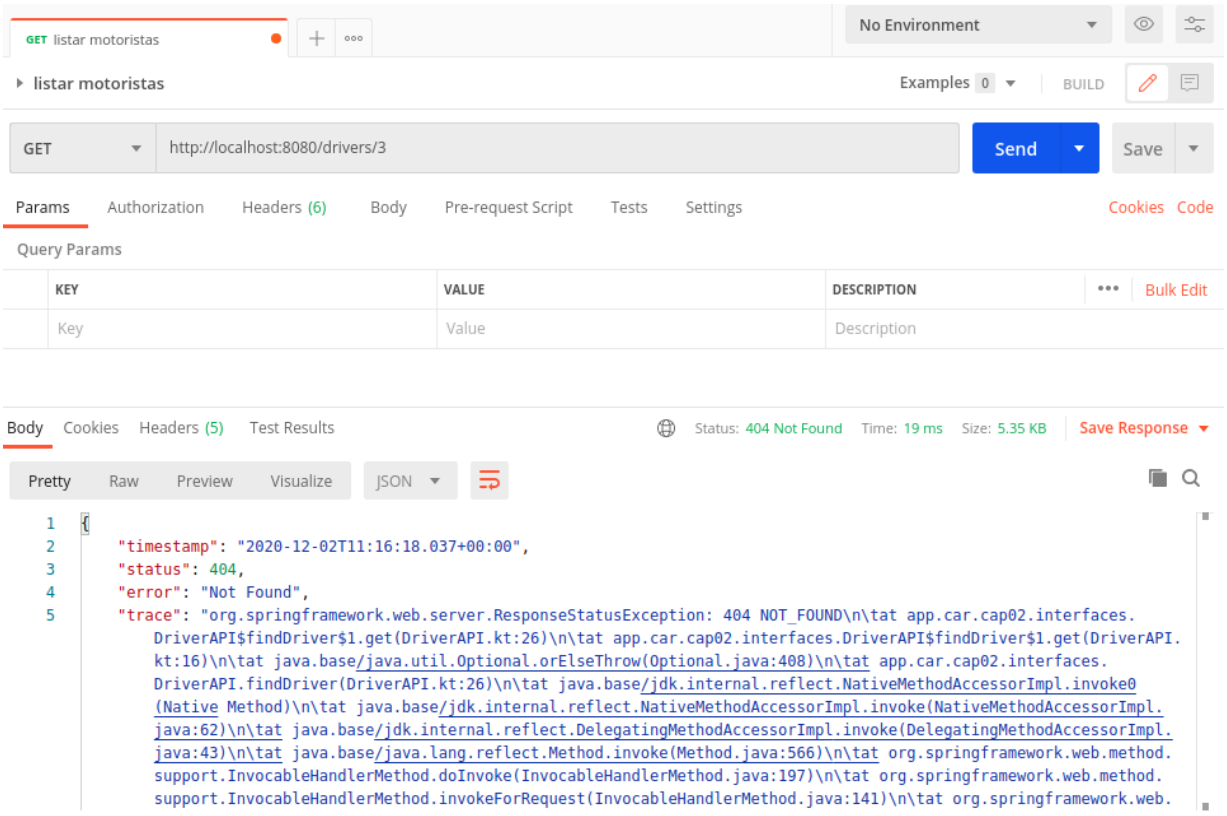


Figura 2.8: Status 404 visto com Postman

Agora, o campo `Status` mostra o texto `404 Not Found`, ou seja, o código 404 continua se fazendo presente. Além disso, você pode observar que agora o corpo da resposta é um JSON, diferentemente de quando consultamos utilizando o *browser*. Por que isso acontece?

2.4 Negociação de conteúdo

Se você reparar bem, verá que pouco abaixo do campo onde é inserida a URL do recurso existe um campo chamado `Headers`:

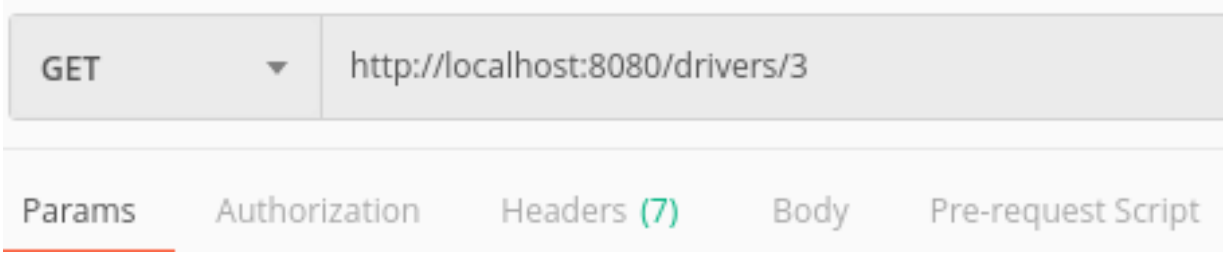


Figura 2.9: O campo Headers logo abaixo da URL

Ao clicar nesse botão, aparece uma seção chamada `Headers` com o número 0 e, ao lado, uma seção chamada `Temporary Headers` com o número 7. Ao expandir a seção `Temporary Headers`, vemos o seguinte:

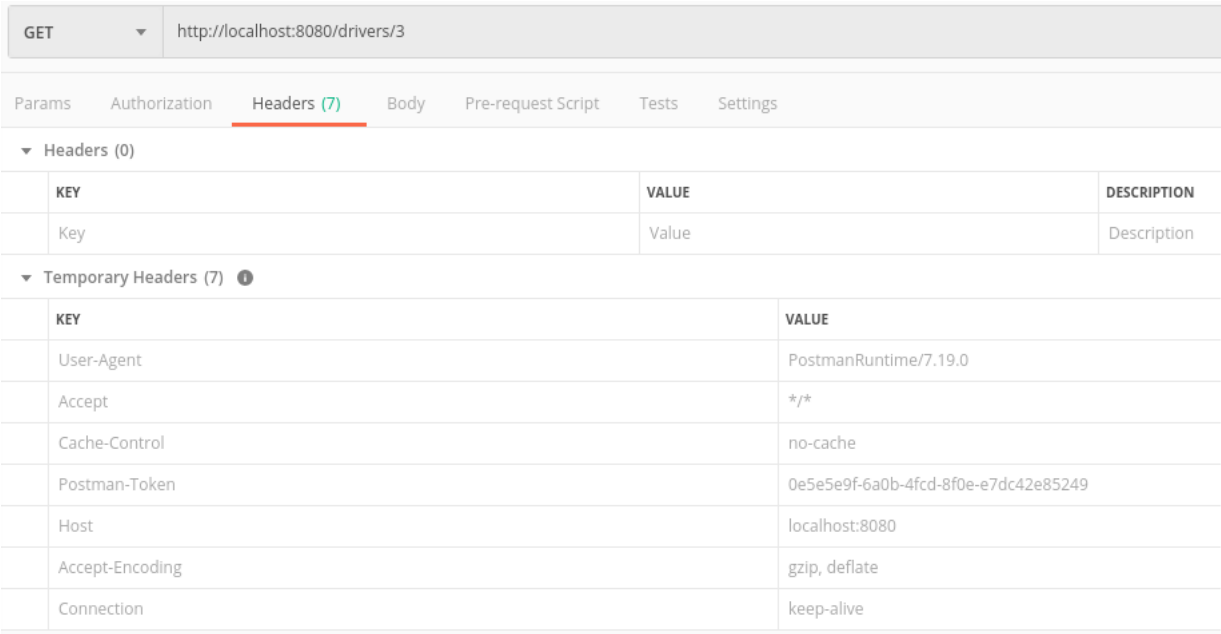


Figura 2.10: Seção de temporary headers expandida

Essa é a seção de cabeçalhos (ou `headers`) que são enviados para o servidor no momento da requisição. Na coluna `Key` existe um cabeçalho chamado `Accept`, que está com o valor `*/*`. Esse cabeçalho é um "pedido" para o servidor enviar o tipo de informação que está destacada no valor; `*/*` significa que o cliente aceita literalmente qualquer coisa. Pode ser uma imagem, HTML, PDF, XML, JSON etc.

Ao fazer isso, o nosso serviço no Spring Boot acaba escolhendo o melhor tipo de informação a enviar, isto é, um JSON descrevendo o problema. No entanto, o *browser* envia algo muito diferente. O valor exato que é enviado pode variar, mas o que capturei no Google Chrome versão 74.0.3729.108, rodando em um Ubuntu Linux 18.04.2, foi o seguinte: `text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3` . Quando esse valor é enviado, o servidor interpreta que o melhor formato a ser utilizado no envio das informações é o `text/html` , um HTML. Mas como essa seleção é feita?

O primeiro passo é segmentar os dados. Observe que acima existem vírgulas separando os valores e, se as retirarmos, vamos enxergar o seguinte:

- `text/html`
- `application/xhtml+xml`
- `application/xml;q=0.9`
- `image/webp`
- `image/apng`
- `*/*;q=0.8`
- `application/signed-exchange;v=b3`

Esses valores são chamados de `MIME Types` , ou alternativamente, `Media Types` (lembra-se de quando utilizamos a anotação `@RequestMapping` na nossa API?). Cada um desses `media types` tem uma estrutura assim: `<tipo>/<subtipo>` . O tipo é como se fosse a "família" das informações, ou seja, texto, imagem etc. Um tipo particular é `application` , que geralmente representa dados binários mas pode ser utilizado em texto estruturado - JSON, por exemplo.

A respeito do `*/*` que temos na listagem, acontece o seguinte: o `*` (asterisco) é utilizado como um curinga nos `media types` . Isso quer dizer que, em uma API que serve imagens, eu posso utilizar o *media type* `image/*` para obter uma imagem genérica de qualquer tipo (JPEG, PNG, GIF etc.). Quando o curinga está dos dois lados, como

apresentado ali em cima, significa "qualquer coisa". O Chrome pediu para o servidor enviar qualquer tipo de dado disponível.

Mas você deve estar se perguntando: "Faz sentido enviar todos esses tipos de dados junto com uma solicitação de 'qualquer coisa'? Por que não enviar somente o curinga, já que o servidor vai enviar qualquer coisa mesmo?". A resposta para essa pergunta é: faz sentido enviar essa solicitação devido ao parâmetro extra que está acompanhando o `*/*`. Observe que o `media type` completo é `*/*;q=0.8`. O ponto e vírgula é usado para separar o `media type` dos parâmetros, e o `q` é um parâmetro que determina a **preferência** por esse tipo de dado (nesse caso, 0.8, em um *range* que varia de 0.1 até 1.0, sendo este último o padrão). Se a preferência por qualquer tipo de dado é diminuída, significa que o servidor vai priorizar a ordem de entrega dos dados assim:

- `text/html`
- `application/xhtml+xml`
- `image/webp`
- `image/apng`
- `application/signed-exchange`
- `application/xml`
- `*/*`

Portanto, ele vai usar o parâmetro `q` para estabelecer uma ordem de prioridade e, depois, por ordem em que os `media types` aparecem.

Vamos fazer um experimento: insira o cabeçalho `Accept` no Postman com o valor `text/html,*/*` e depois aperte `Send`. A resposta deve ser semelhante à seguinte:

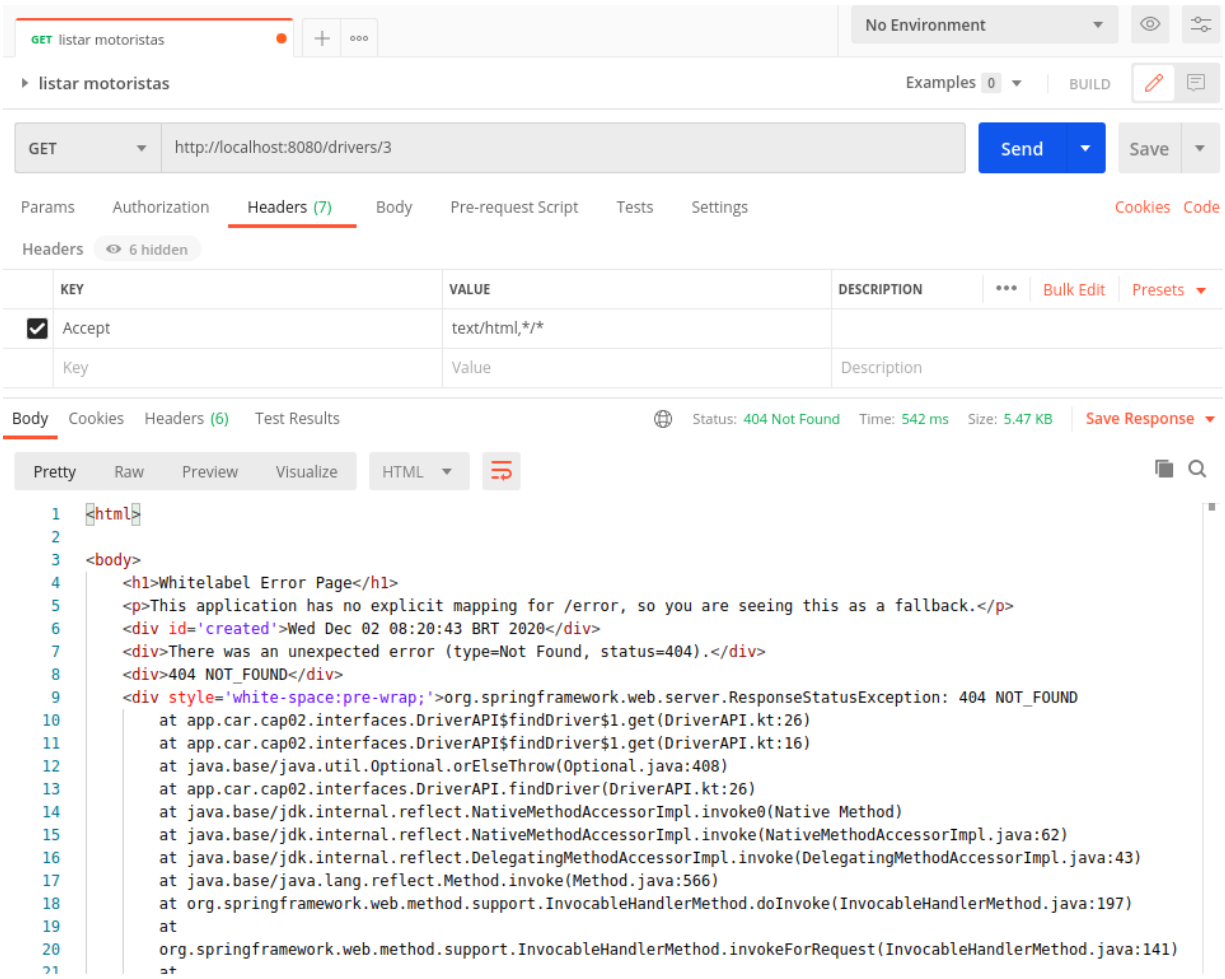


Figura 2.11: A request retorna HTML quando o cabeçalho Accept muda

MAS POR QUE UTILIZAR O TEXT/HTML COM `/` ?

Nos testes que fiz, o Spring Boot apresentou este resultado. Podemos, portanto, considerá-lo um *bug* - o correto seria retornar o HTML mesmo sem o `*/*` no cabeçalho. Veremos a seguir o que acontece se não passarmos esse valor.

Em compensação, colocar nesse cabeçalho um `media type` que não é aceito pelo servidor provoca o retorno do status `406 Not Acceptable`. Se colocarmos `*/*` ou `application/json`, ele retorna o JSON que vimos anteriormente.

Vale observar que, para que o mecanismo de negociação de conteúdo funcione, é necessário que o servidor também informe qual tipo de conteúdo está sendo servido de fato. Isso porque, caso o cliente utilize um curinga (ou mesmo dois, no caso da requisição com `*/*`), é necessário que o servidor avise o que está realmente sendo fornecido. Esse aviso é feito através do cabeçalho `Content-Type`, que retorna sempre um único `media type` e **sem** curinga. No nosso caso, a API sempre retorna JSON, pois foi o que declaramos no cabeçalho `@RequestMapping`. Alguns capítulos adiante veremos como servir outros tipos de conteúdo.

2.5 Enviando dados para o servidor

Agora que já vimos vários aspectos da listagem de motoristas, veremos como fazer a criação de dados utilizando a API.

Como você viu no capítulo passado, a criação de dados do lado do servidor é feita utilizando o método HTTP `POST`. Esse método é especial, pois permite que o usuário utilize o corpo da requisição para passar parâmetros, ao contrário do `GET`, que só permite que o servidor envie dados no corpo.

Vamos modificar nossa API para fazer isso. Primeiro, crie um método na classe `DriverAPI` com a seguinte assinatura:

```
import app.car.cap02.domain.Driver

//...

fun createDriver(driver: Driver) : Driver
```

De maneira semelhante ao método `listDrivers`, vamos anotar nosso método com `@PostMapping`, indicando para o Spring que esse método vai responder ao `POST`. Da mesma forma, vamos passar como parâmetro a URL `/drivers`:

```
import org.springframework.web.bind.annotation.PostMapping;

//...

@PostMapping("/drivers")
fun createDriver(driver: Driver) : Driver
```

> **POR QUE VAMOS USAR A MESMA URL DE ANTES?**
>
> Em REST, a URL de um recurso é apenas um dos aspectos da forma como interagimos com ele. Ela representa a dimensão da nomenclatura e hierarquia do recurso, mas a forma como interagimos com o recurso é determinada pelos métodos HTTP. Assim sendo, a URL permanece sendo a mesma, porque ainda estamos lidando com motoristas.
>
> Criando uma analogia para melhor compreensão, imagine que você tem um par de tênis. Você o compra e o coloca na sua sapateira (`POST`). Você o procura para utilizar (`GET`) várias vezes. Em alguma dessas vezes, você pode retirá-lo para lavar, e devolvê-lo talvez com outras características (`PATCH`). Um dia, você o vende (ou doa) - `DELETE` .
>
> Observe que a interação é com o tênis, mas a forma/intenção pode ser diferente. É por isso que utilizamos a mesma URL: os recursos permanecem os mesmos; o que muda é a forma como lidamos com eles.

Precisamos também informar ao Spring que o `Driver` passado como parâmetro vai estar presente no corpo da requisição. Para isso, utilizamos a anotação `@RequestBody` :

```
import org.springframework.web.bind.annotation.RequestBody;
//...
@PostMapping("/drivers")
fun createDriver(@RequestBody driver: Driver) : Driver
```

Finalmente, utilizamos o método `save` do repositório para salvar o motorista e retornar os dados já persistidos no banco de dados:

```
@PostMapping("/drivers")
fun createDriver(@RequestBody driver: Driver) =
    driverRepository.save(driver)
```

Agora, vamos realizar os testes da nova API com Postman. Ao lado da aba com o nosso método `GET` (lá onde está escrito `listar motoristas`), existe um botão de `+`. Clique nele e, na sequência, vá até o *dropdown* onde está listado o método `GET` e mude para `POST`. Na parte da URL, coloque a mesma de antes, `http://localhost:8080/drivers`. Então, clique no botão `Body` e no *radio* onde está escrito `raw`. Isso quer dizer que o corpo da requisição receberá dados brutos. Um *dropdown* à direita apareceu; mude o valor deste para `JSON`:

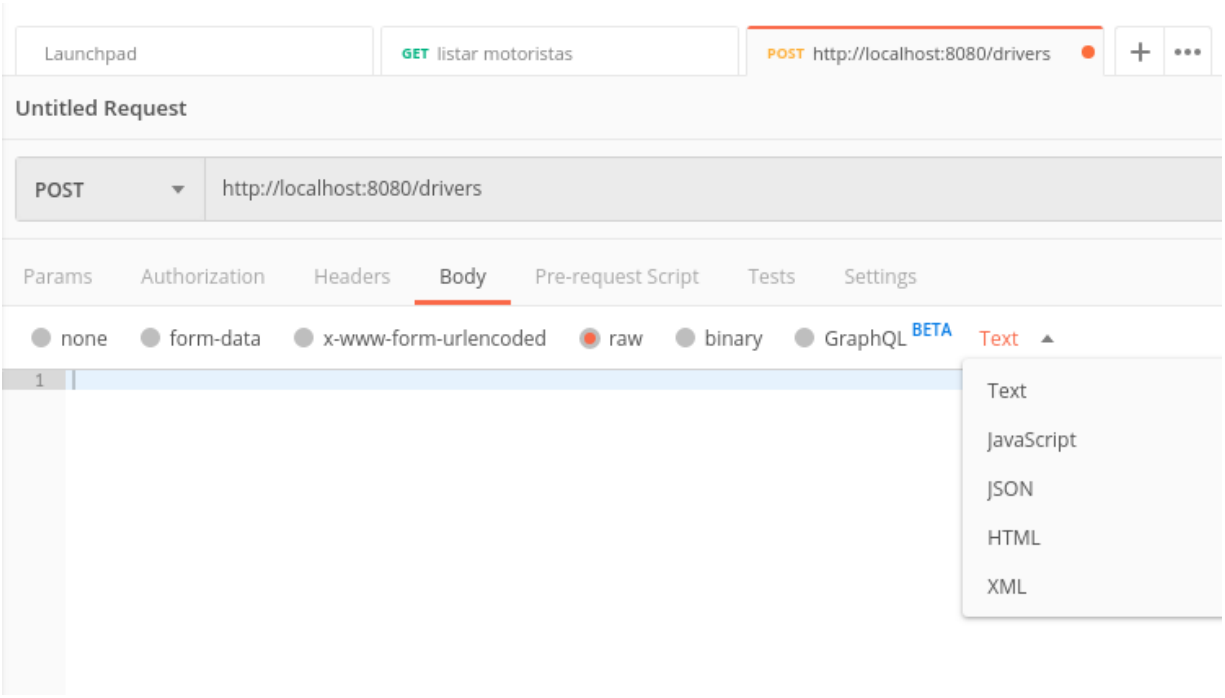


Figura 2.12: Configurando o Postman para realizar a requisição

Vamos preencher agora os dados compatíveis com a requisição. Como vimos antes, trata-se de um JSON com os campos `name` e `birthDate`:

```
{
    "name":"Fulano de Tal",
    "birthDate":"1960-01-01"
}
```

Abro um pequeno interlúdio aqui para fazer uma observação: clique na seção `Headers` e você verá que o Postman criou um cabeçalho `Content-Type` e o preencheu com `application/json`. Observe que, quando enviamos dados, também precisamos enviar o cabeçalho mencionado com o tipo de conteúdo que estamos enviando. Quando selecionamos `JSON` no *dropdown*, o Postman preencheu esse cabeçalho para nós automaticamente.

Voltando ao nosso teste da requisição, apertamos o botão `Send` agora e recebemos a seguinte resposta:

```
"status": 500,
"error": "Internal Server Error",
"message": "ids for this class must be manually assigned before calling
save(): app.car.cap02.domain.Driver; nested exception is
org.hibernate.id.IdentifierGenerationException: ids for this class must be
manually assigned before calling save(): app.car.cap02.domain.Driver"
```

Isso quer dizer que, quando criamos a nossa classe `Driver`, não oferecemos ao Spring/JPA qualquer indicativo de como realizar a geração dos índices no banco de dados. Podemos corrigir isso anotando o campo `id` da classe `Driver` com `@GeneratedValue`:

```
package app.car.cap02.domain

// restante dos imports omitidos
import javax.persistence.GeneratedValue

@Entity
data class Driver(
        @Id
        @GeneratedValue
        var id: Long? = null,
        val name: String,
```

```
        val birthDate: LocalDate
)
```

Reinicie a aplicação e aperte o botão `Send` novamente. O resultado deverá ser o seguinte:

```
{
    "id": 1,
    "name": "Fulano de Tal",
    "birthDate": "1960-01-01"
}
```

2.6 Idempotência: os efeitos de invocações sucessivas

Clique no botão `Send` várias vezes seguidas. Observe que o `id` vai sendo incrementado, ou seja, a cada clique do botão um novo recurso é criado ao lado do servidor. Esse fato levanta uma questão muito importante: suponha que você realize o envio de uma informação para o servidor, mas quando o servidor está enviando a resposta, por algum motivo, a conexão criada é perdida. Dessa forma, você não consegue receber o resultado da invocação e não sabe se a requisição teve sucesso ou não. O que fazer? Reenviar a requisição ou não?

Existe uma propriedade dos métodos HTTP que é a **idempotência**, que trata dos efeitos causados por essas sucessivas invocações ao servidor (da mesma forma que você fez clicando várias vezes sobre o botão `Send`). Dizemos que uma requisição é idempotente quando o efeito da enésima requisição é igual ao da primeira. Em outras palavras, não importa quantas vezes enviamos uma requisição, nenhuma alteração sobre o estado do servidor será produzida, além daquela feita pela primeira, que tinha as mesmas características. Obviamente, falamos de um servidor isolado, isto é, não há nenhum outro cliente interagindo com os mesmos recursos.

Isso nos permite ter uma segurança quando falamos a respeito do cenário em que não recebemos a resposta. Podemos simplesmente reenviar a requisição - se o método é idempotente, o resultado esperado será o mesmo que a primeira requisição. Isso não vale para métodos que não são idempotentes, pois podemos criar dados no servidor além daqueles necessários e criar efeitos indesejados.

Observe também que essa propriedade trata exclusivamente da relação entre o **cliente** e o **servidor**, em termos de recursos. Por exemplo: suponha que eu uso um método idempotente sobre o servidor, mas a cada invocação nova um *log* é gerado ao lado do servidor. Mesmo assim, a requisição é considerada idempotente, porque os *logs* dizem respeito exclusivamente ao comportamento interno do servidor, não fazendo parte da relação cliente e servidor.

Nos exemplos que vimos até agora, o método `GET` é idempotente e o `POST` não. Por esse motivo, inclusive, os *browsers* muitas vezes nos perguntam se queremos reenviar dados: eventualmente estamos preenchendo um formulário, ou algo do tipo, e os dados são enviados usando `POST` . Quando de alguma forma solicitamos que a página seja atualizada (apertando o botão `F5` ou de alguma outra forma), o *browser* pede que o usuário assuma o risco de reenviar os dados, pois isso pode gerar esses efeitos colaterais.

2.7 Atualizando os dados enviados com PUT e PATCH

E por falar em idempotência, temos os casos dos métodos `PUT` e `PATCH` , que são utilizados para fazer atualizações (portanto, escrevem dados) ao lado do servidor mas são idempotentes.

Mas por que temos dois métodos para fazer isso? Porque cada um faz a atualização de uma forma. O `PUT` atualiza o conjunto todo; o `PATCH` , apenas um trecho. Vamos ver rapidamente a diferença antes

de realizar a implementação. Vamos rever o JSON do motorista com o `id` 1:

```
{
    "id": 1,
    "name": "Fulano de Tal",
    "birthDate": "1960-01-01"
}
```

Se nós quisermos realizar uma mudança em todos os dados, é melhor utilizar o `PUT` :

```
PUT /drivers/1
```

```
{
   "name":"Alexandre Saudate",
   "birthDate": "1986-08-18"
}
```

Se eu deixar de enviar um dos dados (`name` ou `birthDate`), o uso do `PUT` vai fazer com que meu serviço entenda que esses dados devem ser descartados. A exceção à regra é o campo `id` , que nesse caso é utilizado na URL por dois motivos: o primeiro, para ser mais adequado à noção de recurso em REST. Se eu fizesse o `PUT` diretamente sobre a URL `/drivers` , isso indicaria que estamos fazendo a atualização da lista toda de motoristas, o que não é o caso.

O segundo motivo é para melhorar a condição de concorrência em que o servidor pode se encontrar. Ao efetuar as atualizações de dados sobre um recurso isolado, fica mais fácil lidar com possíveis atualizações concorrentes, em que mais de um usuário pode tentar criar ou atualizar dados no mesmo recurso ao mesmo tempo.

Já o método `PATCH` vai fazer uma mudança apenas incremental. Se utilizarmos esse método sobre o motorista `Fulano de Tal` , faremos a atualização apenas do campo passado no `PATCH` , veja:

```
PATCH /drivers/1
```

```
{
   "name":"Alexandre Saudate"
}
```

Dessa forma, o campo `birthDate` vai permanecer intocado, com o mesmo valor de antes da requisição com `PATCH`. Isso faz com que cada conteúdo possa ser usado nos mesmos recursos, sendo um mais interessante do que o outro em alguns aspectos.

Vamos à implementação. Como você já viu, a URL vai ser semelhante à de localização de um motorista específico, `/drivers/{id}`. Já o corpo da requisição vai receber o JSON contendo o motorista. Logo, a implementação é como se fosse a junção da implementação do método de localizar um motorista específico com o método de criar um novo motorista. Dessa forma, vamos criar a assinatura do método com esses conceitos e a anotação `@PutMapping`, que vai realizar a ligação desse método com o `PUT`:

```
import org.springframework.web.bind.annotation.PutMapping

@PutMapping("/drivers/{id}")
fun fullUpdateDriver(@PathVariable("id") id:Long, @RequestBody
driver:Driver) : Driver {
// implementação...
}
```

Para a implementação do corpo do método em si, vamos fazer com que o método `fullUpdateDriver` chame o método `findDriver`, pois dessa forma o método vai automaticamente fazer com que o código 404 seja retornado em caso de tentarmos fazer a atualização de um motorista inexistente. Vamos também criar uma cópia do motorista encontrado utilizando o método `copy()`, e então fornecer os dados que foram passados como parâmetro para essa cópia. Finalmente, vamos chamar o método `save` do repositório:

```
@PutMapping("/drivers/{id}")
fun fullUpdateDriver(@PathVariable("id") id:Long, @RequestBody
driver:Driver) : Driver {
```

```
    val foundDriver = findDriver(id)
    val copyDriver = foundDriver.copy(
        birthDate = driver.birthDate,
        name = driver.name
    )
    return driverRepository.save(copyDriver)
}
```

Agora, vamos utilizar novamente o Postman para testar nosso método. Crie uma nova requisição no Postman ajustando o método HTTP para `PUT` e a URL para `http://localhost:8080/drivers/1`. Quanto ao corpo da requisição, ajuste para o seguinte:

```
{
   "name": "Alexandre Saudate",
   "birthDate": "1986-08-18"
}
```

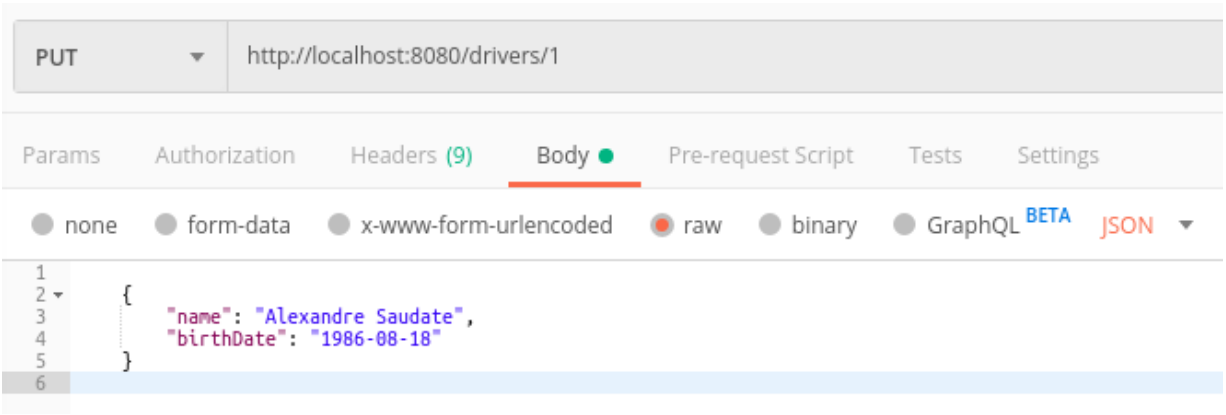


Figura 2.13: Realizando a requisição PUT com Postman

Se tudo estiver certo, você deve receber uma resposta semelhante à seguinte:

```
{
    "id": 1,
    "name": "Alexandre Saudate",
    "birthDate": "1986-08-18"
}
```

Faça o seguinte teste: retire o campo `birthDate` da requisição de entrada, deixando apenas o nome. Em outras palavras, envie o corpo da requisição assim:

```
{
   "name": "Alexandre Saudate",
}
```

O resultado da requisição será semelhante ao seguinte:

{
"timestamp": "2020-12-02T11:53:21.096+00:00",
"status": 400,
"error": "Bad Request",
"trace": "org.springframework.http.converter.HttpMessageNotReadableException: JSON parse error: Instantiation of [simple type, class app.car.cap02.domain.Driver] value failed for JSON property birthDate due to missing (therefore NULL) value for creator parameter birthDate which is a non-nullable type; nested exception is com.fasterxml.jackson.module.kotlin.MissingKotlinParameterException: Instantiation of [simple type, class app.car.cap02.domain.Driver] value failed for JSON property birthDate due to missing (therefore NULL) value for creator parameter birthDate which is a non-nullable type\n at [Source: (PushbackInputStream); line: 3, column: 1] (through reference chain: app.car.cap02.domain.Driver[\"birthDate\"])\n\tat org.springframework.http.converter.json.AbstractJackson2HttpMessageConverter.readJavaType(AbstractJackson2HttpMessageConverter.java:285)\n\tat org.springframework.http.converter.json.AbstractJackson2HttpMessageConverter.read(AbstractJackson2HttpMessageConverter.java:243)\n\tat org.springframework.web.servlet.mvc.method.annotation.AbstractMessageConverterMethodArgumentResolver.readWithMessageConverters(AbstractMessageConverterMethodArgumentResolver.java:205)\n\tat org.springframework.web.servlet.mvc.method.annotation.RequestResponseBodyMethodProcessor.readWithMessageConverters(RequestResponseBodyMethodProcessor.java:158)\n\tat org.springframework.web.servlet.mvc.method.annotation.RequestResponseBodyMethodProcessor.resolveArgument(RequestResponseBodyMethodProcessor.java:131)\n\tat org.springframework.web.method.support.HandlerMethodArgumentResolverComposite.resolveArgument(HandlerMethodArgumentResolverComposite.java:121)\n\tat org.springframework.web.method.support.InvocableHandlerMethod.getMethodArgumentValues(InvocableHandlerMethod.

Figura 2.14: Resposta de erro do Kotlin

Ocorreu um erro com status 400, o que indica que algo na requisição não está no formato correto. Para o nosso caso em específico, indica que o parâmetro `birthDate` não pode ser nulo. Esse erro ocorreu porque criamos a classe `Driver` com o campo `birthDate` como não nulo (quando colocamos um sinal de interrogação próximo ao tipo do campo, informamos ao Kotlin que este campo pode ser nulo - o que não foi o caso com o campo `birthDate`). Dessa forma, ao tentar instanciar o parâmetro para o nosso método, essa falha foi identificada.

Vamos à implementação do método `PATCH`. A principal diferença entre esse método e o método `PUT` é que aquele deve aceitar campos nulos na requisição. Dessa forma, precisamos criar outra classe que possa representar a entrada de dados nesse método. Assim, vamos criar a `data class PatchDriver`:

```
data class PatchDriver(
        val name: String?,
        val birthDate: LocalDate?
)
```

Vamos criar um método com assinatura semelhante à do método `fullUpdateDriver`, mas agora vamos utilizar a classe `PatchDriver` na assinatura, além da anotação `@PatchMapping`:

```
import org.springframework.web.bind.annotation.PatchMapping

@PatchMapping("/drivers/{id}")
fun incrementalUpdateDriver(@PathVariable("id") id:Long, @RequestBody
driver:PatchDriver) : Driver {
//...
}
```

A implementação do método também é semelhante. A diferença é que vamos testar os campos a serem mapeados para que, caso passemos algum dado nulo, a implementação considere o que já havia antes. Podemos fazer isso dentro do método `copy` utilizando o **Elvis operator**, que é uma instrução do Kotlin que indica que, caso o que esteja à esquerda do operador seja nulo, deve ser utilizado o que está à direita:

```
@PatchMapping("/drivers/{id}")
fun incrementalUpdateDriver(@PathVariable("id") id:Long, @RequestBody
driver:PatchDriver) : Driver {
    val foundDriver = findDriver(id)
    val copyDriver = foundDriver.copy(
        birthDate = driver.birthDate ?: foundDriver.birthDate,
        name = driver.name ?: foundDriver.name
    )
    return driverRepository.save(copyDriver)
}
```

Agora, vamos enviar o JSON com o método `PATCH` já sem o campo `birthDate`:

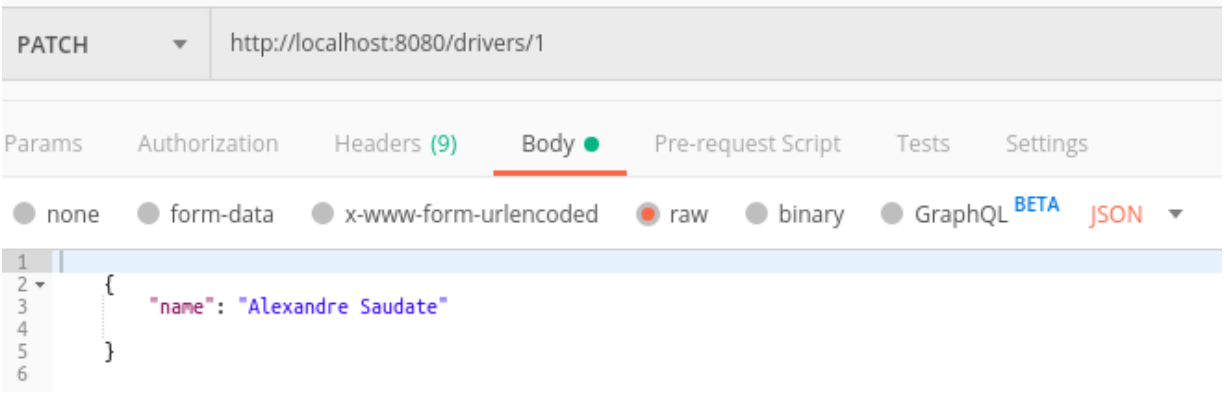


Figura 2.15: Realizando a requisição PATCH com Postman

O resultado deve ser algo como o seguinte JSON:

```
{
   "id": 1,
   "name": "Alexandre Saudate",
   "birthDate": "1960-01-01"
}
```

2.8 Apagando os dados de um determinado motorista

Finalmente, para completar o conjunto das operações básicas que podemos realizar sobre um recurso, vamos falar do método `DELETE`. Deixei esse método por último, porque ele apresenta algumas polêmicas.

A primeira polêmica é em relação ao corpo da requisição. A especificação formal do método `DELETE` não deixa claro se é permitido ou não enviar dados no corpo, o que pode gerar problemas de acordo com a ferramenta utilizada. Algumas ferramentas podem aceitar, outras podem rejeitar por completo (sem aviso) e outras podem retornar um erro HTTP. O Spring Boot, na versão utilizada na escrita deste livro, utiliza um servidor Tomcat para servir as requisições. Esse servidor suporta corpo para o

método HTTP `DELETE`. Eu, no entanto, não recomendo que você o utilize devido aos problemas mencionados.

A segunda polêmica é em relação à idempotência. O método `DELETE` é considerado idempotente, porque apenas a primeira requisição vai produzir alterações sobre o estado do servidor. As requisições subsequentes estarão impossibilitadas de produzir alterações, porque o recurso já terá sido apagado. No entanto, enquanto a resposta da primeira requisição pode ser um `200`, as respostas das requisições seguintes podem ser `200` ou `404` (ou seja, recurso não encontrado), dependendo da implementação. A idempotência, nesse caso, é considerada porque **não foi produzida alteração no estado do servidor**, isto é, a primeira requisição apagou o recurso e, da segunda em diante, o recurso já não existia mais.

Dito isso, penso o seguinte em relação aos códigos retornados da segunda requisição em diante: não há certo ou errado. Depende do cenário de uso - se faz sentido para os possíveis clientes da API receberem uma resposta de erro em caso de reenvio de uma requisição ou não.

Postas essas considerações, vamos à implementação do método no servidor. Assim como os outros métodos HTTP mencionados, a implementação do `DELETE` segue a mesma linha de simplicidade. Basta utilizar a anotação `@DeleteMapping` para realizar o mapeamento, e o método `deleteById` do repositório para efetivamente excluir o dado do repositório. O código fica assim:

```
import org.springframework.web.bind.annotation.DeleteMapping

@DeleteMapping("/drivers/{id}")
fun deleteDriver(@PathVariable("id") id: Long) =
    driverRepository.deleteById(id)
```

Agora, vamos reinicializar o sistema e abrir uma nova aba no Postman para fazer o teste da API. Não se esqueça de que, como nosso banco está em memória, os dados se perdem quando o sistema é reinicializado, então precisamos fazer a criação do

motorista novamente com `POST`. Feito isso, podemos configurar o Postman para enviar uma requisição com o método `DELETE`:

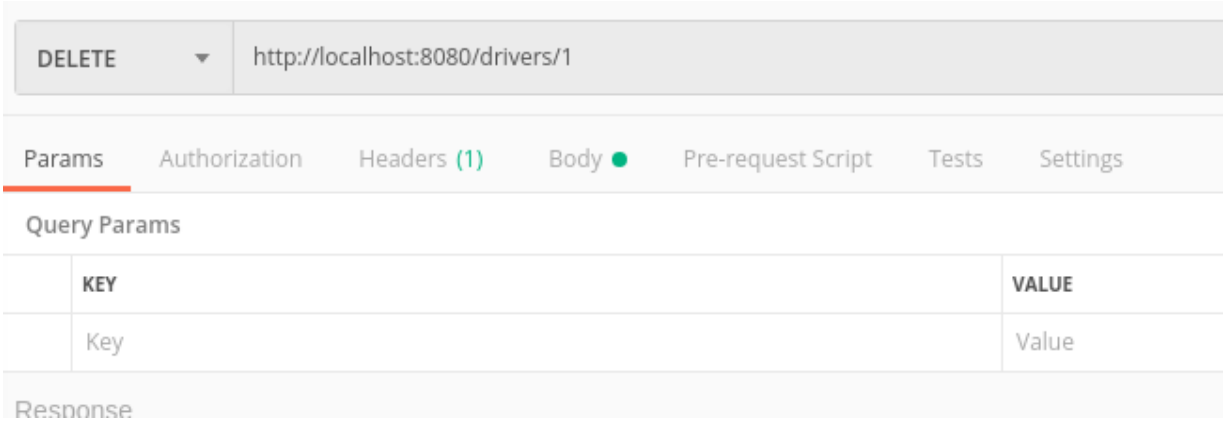


Figura 2.16: Realizando a requisição DELETE com Postman

Ao realizarmos essa requisição, ela retorna o código `200`, o que significa que conseguimos apagar com sucesso o registro. É possível realizar a verificação utilizando o método `GET`.

COMO FAÇO PARA CRIAR UMA API DE DELETE LÓGICO?

O `delete` lógico consiste em ajustar um atributo na entidade para que ela esteja apenas desativada, e o recurso não seja mais listado com o método `GET`. Desde que realmente não seja mais listado, o mapeamento deve ser feito da mesma forma como funciona um `delete` físico, ou seja, utilizando o método HTTP `DELETE` em conjunto com a URL do recurso.

Será feito de outra forma apenas se o recurso puder voltar a ser listado. Por exemplo, se houver alguma forma de trazer o recurso apagado logicamente de volta à listagem, então na verdade deveria ser utilizado o método `PATCH` ou `PUT`. Observe que isso muda inclusive a forma de encarar o recurso, pois ele passa a ser visto como **desativado**, e não **apagado**.

Repare, no entanto, que se tentarmos apagar um motorista que não existe (ou que já tenha sido apagado) recebemos o seguinte erro:

```
{
    "timestamp": "2019-11-08T23:33:48.318+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "No class app.car.cap02.domain.Driver entity with id 1
exists!",
    etc..
}
```

Isso ocorre porque nosso recurso não está realizando o tratamento adequado no caso de o motorista já ter sido apagado. Caso queiramos tratar isso é simples - basta utilizar novamente o método `findDriver` e passar o motorista encontrado como parâmetro para o método `delete` do repositório. Caso o motorista não seja encontrado, o método vai devolver o status `404 Not Found`:

```
@DeleteMapping("/drivers/{id}")
fun deleteDriver(@PathVariable("id") id: Long) =
    driverRepository.delete(findDriver(id))
```

Conclusão

Neste capítulo, conhecemos melhor outros métodos HTTP. Conseguimos interagir de forma mais ampla com o recurso realizando criação de novos recursos, atualização e exclusão. Também conhecemos alguns códigos de status novos e tipos de mídia. Aos poucos, estamos conhecendo melhor a estrutura de um recurso REST e seus pilares - a forma de modelar URLs, quais métodos utilizar, tipos de mídia, códigos de status etc.

Ainda há recursos poderosos a serem conhecidos, estamos apenas começando. Vamos começar a investigar mais profundamente os tipos de mídia e a interação entre os recursos.

Capítulo 3
Criando relacionamentos entre recursos

"O começo é a parte mais importante do trabalho." - Platão.

Agora que já temos uma API para executar as operações de motoristas, vamos seguir rumo à conclusão do nosso primeiro caso de uso. Precisamos criar mais duas APIs: uma que faça a mesma operação para passageiros (ou seja, um cadastro) e uma que receba deles os pedidos de viagens para realizar a interligação entre motoristas e passageiros.

3.1 Criando a API de passageiros

Como já sabemos criar uma API simples, vamos repetir o procedimento de criação de motoristas na plataforma para realizar também o cadastro de passageiros. Vamos criar a entidade que representa os passageiros da seguinte forma (ainda no arquivo `Entities.kt`, abaixo da `data class Driver`):

```
@Entity
data class Passenger(
        @Id
        @GeneratedValue
        var id: Long? = null,
        val name: String
)
```

Vamos também criar o repositório equivalente no arquivo `Repositories.kt`:

```
interface PassengerRepository: JpaRepository<Passenger, Long>
```

Finalmente, vamos criar a API com base no que vimos da API de motoristas:

```
package app.car.cap03.interfaces

import app.car.cap03.domain.*
import org.springframework.http.HttpStatus
import org.springframework.http.MediaType
import org.springframework.stereotype.Service
import org.springframework.web.bind.annotation.*
import org.springframework.web.server.ResponseStatusException

@Service
@RestController
@RequestMapping(path = ["/passengers"], produces =
[MediaType.APPLICATION_JSON_VALUE])
class PassengerAPI(
        val passengerRepository: PassengerRepository
) {

    @GetMapping
    fun listPassengers() = passengerRepository.findAll()

    @GetMapping("/{id}")
    fun findPassenger(@PathVariable("id") id: Long) =
        passengerRepository.findById(id)
            .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }

    @PostMapping
    fun createPassenger(@RequestBody passenger: Passenger) =
        passengerRepository.save(passenger)

    @PutMapping("/{id}")
    fun fullUpdatePassenger(@PathVariable("id") id: Long,
            @RequestBody passenger: Passenger): Passenger {

        val newPassenger = findPassenger(id).copy(
                name = passenger.name
        )
        return passengerRepository.save(newPassenger)
    }

    @PatchMapping("/{id}")
    fun incrementalUpdatePassenger(@PathVariable("id") id: Long,
```

```
            @RequestBody passenger: PatchPassenger): Passenger {

        val foundPassenger = findPassenger(id)
        val newPassenger = foundPassenger.copy(
                name = passenger.name ?: foundPassenger.name
        )
        return passengerRepository.save(newPassenger)
    }

    @DeleteMapping("/{id}")
    fun deletePassenger(@PathVariable("id") id: Long) =
        passengerRepository.delete(findPassenger(id))

}

data class PatchPassenger(
        val name: String?
)
```

Observe que nessa API existe uma pequena diferença em relação à API de motoristas criada anteriormente: a anotação `@RequestMapping` já possui o atributo `path` que, por sua vez, é espelhado para todas as anotações de mapeamento REST da classe. Isso quer dizer que, onde a `URL` REST não é mencionada (no caso, somente no método `listPassengers`), atribui-se a URL `/passengers` a ele. Se a URL for mencionada, então assume-se que a URL `/passengers` vai no começo e o restante, no final. Por exemplo, no método `findPassenger`, toma-se a URL `/passengers` no começo e o `/{id}` (proveniente da anotação `@GetMapping`) no final, sendo que a URL final fica `/passengers/{id}`.

3.2 Criando a API de solicitação de viagens

A próxima API que criaremos apresenta um desafio um pouco diferente do que vimos anteriormente. Aqui, precisamos criar uma API na qual seja possível que um determinado passageiro faça uma

solicitação de viagem. Nesse caso, perceba que não há mais uma simples requisição para um banco de dados sendo feita, mas sim uma ação a ser tomada. É aqui onde precisamos começar a adotar algumas técnicas específicas de modelagem para não criar errado.

Antes de mais nada, precisamos primeiro conhecer qual é o recurso específico com o qual estamos lidando. Pode parecer bobagem, mas já vi muitos desenvolvedores experientes errarem em modelagem de APIs porque não sabiam com qual recurso estavam lidando. Portanto, tenha em vista que, em um primeiro momento, estamos lidando com **solicitações de viagens**, o que é diferente de **viagens**. O primeiro recurso, ao haver aceite de um motorista, é convertido no segundo.

O primeiro passo é modelar a URL. Pode parecer tentador modelar um em função do outro; no entanto, isso é uma armadilha. Como já mencionei antes, as URLs dos recursos REST são definidas de forma hierárquica. Isso quer dizer que, se pegarmos uma URL e separarmos cada porção pela `/`, as porções mais à direita serão encaradas como subconjuntos das porções que ficarem mais à esquerda. Por exemplo, o site `https://restfulapi.net/`, na seção `REST Resource Naming Guide` (`Guia para nomenclatura de recursos REST`), diz o seguinte (em tradução livre):

> Um recurso pode conter subcoleções de recursos. Por exemplo, o recurso subcoleção "contas" de um "cliente" em particular pode ser identificado pela URI `/customers/{customerId}/accounts` (em um domínio bancário). De forma similar, um recurso singleton "conta" dentro do recurso subcoleção "contas" pode ser identificado como se segue:
>
> ```
> /customers/{customerId}/accounts/{accountId}
> ```

Isso quer dizer que podemos interpretar as URLs como conjuntos e cada um dos trechos seguintes (separados por `/`) pode ser interpretado como subconjunto do conjunto principal.

No entanto, no nosso contexto em particular, é um erro considerar que existe tal relação entre as solicitações de viagens e as viagens. Vamos analisar o motivo:

- Se modelarmos a URL como `/solicitacoes/viagens`, vamos acabar considerando que todas as viagens pertencem de uma vez só a todas as solicitações de viagens, o que é um erro.
- Se modelarmos a URL como `/solicitacoes/{id}/viagens`, vamos acabar chegando à conclusão de que sempre vamos precisar do ID da solicitação de viagem para conseguir chegar à viagem desejada, o que prejudica a usabilidade da nossa API.
- Se modelarmos a URL como `/viagens/solicitacoes`, temos um problema semelhante ao primeiro caso.
- Se modelarmos a URL como `/viagens/{id}/solicitacoes`, não vamos conseguir criar uma solicitação sem antes ter uma viagem, o que é um erro semântico.

O próximo passo é identificar como será o recurso de solicitação de viagens. Para realizar uma solicitação, precisamos:

- do passageiro;
- do local de origem;
- do local de destino.

Com esses dados, vamos criar a entidade de solicitação de viagens:

```
@Entity
data class TravelRequest (
        @Id
        @GeneratedValue
        var id: Long? = null,

        @ManyToOne
        val passenger: Passenger,
        val origin: String,
        val destination: String
)
```

Agora, vamos criar um repositório para essa entidade:

```
public interface TravelRequestRepository extends
JpaRepository<TravelRequest, Long> {}
```

Finalmente, vamos criar a classe de API. Diferente das outras, vamos inicialmente criar apenas uma "casca" dessa API, contendo apenas a parte de criação do recurso de solicitação de viagem:

```
@Service
@RestController
@RequestMapping(path = ["/travelRequests"], produces =
[MediaType.APPLICATION_JSON_VALUE])
class TravelRequestAPI {

    @PostMapping
    fun makeTravelRequest(@RequestBody travelRequest: TravelRequest) {

    }
}
```

3.3 Criação do serviço de solicitação de viagens

Como você pôde observar, a API que criamos para fazer as solicitações de viagens está vazia. Isso porque temos os seguintes passos a realizar:

- Salvar as solicitações de viagens.
- Fornecer uma API onde os motoristas interessados possam obter viagens.
- Mas somente devolver para esses motoristas interessados as viagens que geograficamente estiverem próximas deles.

Vamos então criar a classe `TravelService`, que vai desempenhar a função de chamar o repositório para salvar o pedido:

```
package app.car.cap03.domain

import org.springframework.stereotype.Component
```

```
@Component
class TravelService(
    val travelRequestRepository: TravelRequestRepository
) {

    fun saveTravelRequest(travelRequest: TravelRequest) =
        travelRequestRepository.save(travelRequest)
}
```

Com isso, já podemos fazer a `TravelRequestAPI` ter uma referência para `TravelService` e salvar a solicitação de viagem:

```
class TravelRequestAPI(
    val travelService: TravelService
) {

    @PostMapping
    fun makeTravelRequest(@RequestBody travelRequest: TravelRequest) {
        travelService.saveTravelRequest(travelRequest)
    }
}
```

O que falta agora é determinar o que será retornado para o usuário. Diferentemente de outros recursos, este é um que deve sofrer uma mudança com o passar do tempo, ou seja, uma resposta para o cliente pode ser dada nesse momento, mas essa é uma resposta apenas intermediária. No nosso caso de uso específico, a resposta definitiva deve vir quando algum motorista aceitar a viagem, o que costuma ser comum quando há o uso de mensageria e requisições assíncronas.

Esse é um caso em que costumamos retornar o código HTTP `202 Accepted`. Esse código nos obriga a retornar um cabeçalho `Location`, que o cliente pode usar para consultar o resultado definitivo da requisição. Isso significa que precisamos agora ter uma API onde seja possível consultar o status da requisição e, portanto, essa requisição precisa *ter* um status. No entanto, perceba que esse status só seria necessário na saída da API, e não na requisição. Em

outras palavras, se eu colocar um campo chamado `status` na `TravelRequest` apenas com a finalidade de deixar isso disponível na saída da minha API, ele estaria acessível também na requisição, apesar de desnecessário. Como corrigir isso?

A forma ideal de corrigir seria através do *design pattern* `Data Transfer Object`. Ele consiste na criação de objetos que servirão de intermediários na transmissão de dados, o que é exatamente o que precisamos. Assim sendo, vamos criar as classes que farão a representação da entrada de dados da nossa API e também da saída. Vamos nomear a classe de entrada de `TravelRequestInput`:

```
data class TravelRequestInput(
    val passenger: Passenger,
    val origin: String,
    val destination: String
)
```

Observe que nessa classe temos também uma referência ao objeto `Passenger`, sendo que na verdade precisaríamos apenas do `id` dele. Fazendo a substituição, temos a classe:

```
data class TravelRequestInput(
    val passengerId: Long,
    val origin: String,
    val destination: String
)
```

Podemos fazer a substituição na nossa API:

```
@PostMapping
fun makeTravelRequest(@RequestBody travelRequestInput: TravelRequestInput)
{
    travelService.saveTravelRequest(travelRequestInput)
}
```

Agora, temos um outro problema: a `travelRequestInput` não é compatível com o `travelService`. Não podemos fazer a `travelService` passar a aceitar a `TravelRequestInput`, pois isso feriria os princípios da arquitetura limpa. Ou seja, precisamos fazer o mapeamento das

entidades nessa mesma camada. Para isso, vamos criar uma classe chamada `TravelRequestMapper`. Essa classe vai definir um método que faça o mapeamento do `TravelRequestInput` para um objeto do tipo `TravelRequest`, inclusive procurando pelo passageiro e lançando um código `404` caso não o encontre:

```
package app.car.cap03.interfaces.mapping

//imports omitidos

@Component
class TravelRequestMapper(
    val passengerRepository: PassengerRepository
) {

    fun map(input: TravelRequestInput) : TravelRequest {

        val passenger = passengerRepository.findById(input.passengerId)
            .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }

        return TravelRequest(passenger = passenger,
            origin = input.origin,
            destination = input.destination)
    }
}
```

Vamos modificar nossa classe de API para utilizar o componente de mapeamento antes de repassar os dados para a classe de serviço:

```
class TravelRequestAPI(
    val travelService: TravelService,
    val mapper: TravelRequestMapper
) {

    @PostMapping
    fun makeTravelRequest(@RequestBody travelRequestInput:
TravelRequestInput) {
        travelService.saveTravelRequest(mapper.map(travelRequestInput))
    }
}
```

E criar um enum que represente os possíveis status para as solicitações, tais como: solicitação criada, aceita ou recusada:

```
enum class TravelRequestStatus {
        CREATED, ACCEPTED, REFUSED
}
```

Vamos agora vincular esse `enum` à nossa classe, e aproveitar para criar um campo que represente a data de criação da solicitação. Observe que esses campos serão criados e automaticamente populados com os valores padrão `CREATED` e a data atual. Dessa forma, caso eles não sejam explicitamente fornecidos no construtor da classe, o Kotlin utilizará esses valores como os padrões:

```
import java.time.LocalDateTime

@Entity
data class TravelRequest (
        @Id
        @GeneratedValue
        var id: Long? = null,

        @ManyToOne
        val passenger: Passenger,
        val origin: String,
        val destination: String,
        val status: TravelRequestStatus = TravelRequestStatus.CREATED,
        val creationDate: LocalDateTime = LocalDateTime.now()
)
```

3.4 Inserindo links: primeiro uso de HATEOAS

Vamos agora criar uma classe para representar a saída da nossa API, da mesma forma como fizemos com a entrada. Vamos começar copiando os atributos da classe `TravelRequest`, tirando as anotações e os valores que são populados automaticamente:

```
data class TravelRequestOutput(
    val id: Long,
    val passenger: Passenger,
    val origin: String,
    val destination: String,
    val status: TravelRequestStatus,
    val creationDate: LocalDateTime
)
```

Observe a presença da classe `Passenger`. Essa classe, por si só, é tratada em uma API separada. Se utilizarmos a classe `TravelRequestOutput` da forma como está, o passageiro como um todo será referenciado na resposta - algo que talvez não seja desejado. Podemos tratar esse caso utilizando uma técnica chamada HATEOAS.

HATEOAS é a abreviação para **Hypermedia As The Engine Of Application State**, ou Hipermídia como o motor de estado da aplicação. Esse é um nome grande para indicar a noção de que REST deve utilizar links para relacionar recursos conectados entre si.

Pense em uma página da web. Ao acessar uma página, diversos recursos são carregados de formas distintas. A página em si vem de uma vez só, mas contendo diversos links para JavaScript, CSS e imagens relacionadas. Todos esses recursos são carregados ao final, mas seguir com essa estratégia tem várias vantagens, como o *cache* separado dos recursos e a possibilidade de fazer a carga de forma paralela ou mesmo sob demanda.

Da mesma forma, em REST utilizamos esses conceitos para demonstrar a relação entre recursos que estão conectados entre si, mas que não pertencem necessariamente um ao outro. Quando queremos demonstrar essa relação, utilizamos links.

O Spring facilita a criação desses links com o subprojeto `Spring HATEOAS`. Para utilizá-lo, vamos adicionar a seguinte dependência no nosso `build.gradle.kts`:

```
implementation("org.springframework.hateoas:spring-hateoas")
```

O próximo passo é alterar a classe `TravelRequestOutput` para que ela não mais tenha uma referência para `Passenger`. Vamos usar o Spring HATEOAS para construir essa referência fora dessa classe, de forma que ela passe a ter a seguinte estrutura:

```
data class TravelRequestOutput(
    val id: Long,
    val origin: String,
    val destination: String,
    val status: TravelRequestStatus,
    val creationDate: LocalDateTime
)
```

Vamos agora construir um método na classe `TravelRequestMapper` que faça o mapeamento do resultado do processamento contido na classe `TravelRequest` para a classe `TravelRequestOutput`. Esse método vai fazer apenas uma transposição dos valores:

```
fun map(travelRequest: TravelRequest) : TravelRequestOutput {
    return TravelRequestOutput(
        id = travelRequest.id!!,
        origin = travelRequest.origin,
        destination = travelRequest.destination,
        status = travelRequest.status,
        creationDate = travelRequest.creationDate
    )
}
```

Observe que a passagem do campo `id` precisa conter ao final duas exclamações (`!!`), que é uma forma de informar ao compilador que, apesar de esse campo poder ser nulo, nesse contexto em que está sendo usado, ele nunca poderá ser nulo - e se for, deve lançar uma exceção.

Agora é onde vamos começar a usar o Spring HATEOAS para construir nossa estrutura. O Spring HATEOAS nos fornece uma hierarquia de classes para lidar com recursos de vários tipos:

- Para trabalhar com links de uma única entidade, a melhor classe a ser utilizada é a `org.springframework.hateoas.EntityModel`.
- Para trabalhar com links sobre uma coleção de entidades, a melhor classe a ser utilizada é a `org.springframework.hateoas.CollectionModel`.
- Para trabalhar com links sobre uma coleção com suporte a paginação, a melhor classe a ser utilizada é a `org.springframework.hateoas.PagedModel`.
- Caso nenhuma das classes acima atenda às necessidades do projeto, é possível criar uma extensão da classe `org.springframework.hateoas.RepresentationModel` (que todas as classes listadas acima já estendem).

Conforme visto, podemos assumir que a melhor classe a ser utilizada para retornar o link com a referência ao passageiro é a `EntityModel`. Assim, vamos criar mais um método na classe `TravelRequestMapper` que receba tanto o `TravelRequest` quanto o `TravelRequestOutput`, que foi construído pelo método `map`. Dessa forma, esse método deverá retornar um objeto do tipo `EntityModel`, que é criado pelo método estático `of()`:

```
fun buildOutputModel(travelRequest: TravelRequest, output:
TravelRequestOutput) =
    EntityModel.of(output)
```

O último ponto a ser construído é a representação do link em si. Para isso, no entanto, temos que entender como um link funciona.

Um link HATEOAS tem a mesma forma que um link em uma página da web. Isso quer dizer que existem alguns atributos básicos que ele precisa definir, como qual tipo de relação ele apresenta com o recurso atual e alguns atributos opcionais, como um título explicativo do que aquele link contém. Além disso, obviamente, ele precisa da URL para onde aquele link leva.

O Spring HATEOAS nos fornece uma classe que dá assistência para criação de links, a `org.springframework.hateoas.server.mvc.WebMvcLinkBuilder`. Essa classe

consegue referenciar classes de APIs, como a `PassengerAPI`, de maneira que, caso mudemos a URL da API referenciada, os links recebem a atualização automaticamente. Isso é feito através do método `linkTo` passando a classe da API como parâmetro:

```
val passengerLink = WebMvcLinkBuilder.linkTo(PassengerAPI::class)
```

No entanto, esse código não compila. Ele continua referenciando o `builder`, indicando que ainda existem alguns dados que precisamos fornecer. O próximo dado é o `rel`, ou seja, a construção de um atributo do link que indica o que está sendo representado - no nosso caso, um passageiro. Vamos então utilizar o método `withRel` passando o `passenger` como parâmetro:

```
val passengerLink = WebMvcLinkBuilder
    .linkTo(PassengerAPI::class)
    .withRel("passenger")
```

Da forma como está, ele já compila. Mas seria interessante acrescentar apenas um atributo explicativo a esse link, para que o cliente possa decidir facilmente se precisa seguir o link ou não. Esse atributo explicativo é chamado `title`, e um bom valor para ele seria o nome do passageiro. Vamos utilizar o método `withTitle` para fazer essa associação:

```
val passengerLink = WebMvcLinkBuilder
    .linkTo(PassengerAPI::class)
    .withRel("passenger")
    .withTitle(travelRequest.passenger.name)
```

Finalmente, nosso link está construído. Agora, basta adicioná-lo ao modelo que construímos antes. O método `buildOutputModel` fica assim:

```
fun buildOutputModel(travelRequest: TravelRequest, output:
TravelRequestOutput):
    EntityModel<TravelRequestOutput> {

    val passengerLink = WebMvcLinkBuilder
        .linkTo(PassengerAPI::class)
```

```
        .withRel("passenger")
        .withTitle(travelRequest.passenger.name)

    return EntityModel.of(output, passengerLink)
}
```

Agora, precisamos alterar a classe de API. Para isso, temos que realizar o seguinte:

1. Primeiro, temos que alterar o tipo de retorno (para retornar o `EntityModel`).
2. Depois, temos que modificar o método para construir um `TravelRequestOutput` (através do método `map`).
3. Construir um `EntityModel` (através do método `buildOutputModel`).

O código com essas alterações fica assim:

```
@PostMapping
fun makeTravelRequest(@RequestBody travelRequestInput: TravelRequestInput)
    : EntityModel<TravelRequestOutput> {
    val travelRequest =
travelService.saveTravelRequest(mapper.map(travelRequestInput))
    val output = mapper.map(travelRequest)
    return mapper.buildOutputModel(travelRequest, output)
}
```

Finalmente, vamos testar nossa API. Primeiro, vamos recriar o passageiro:

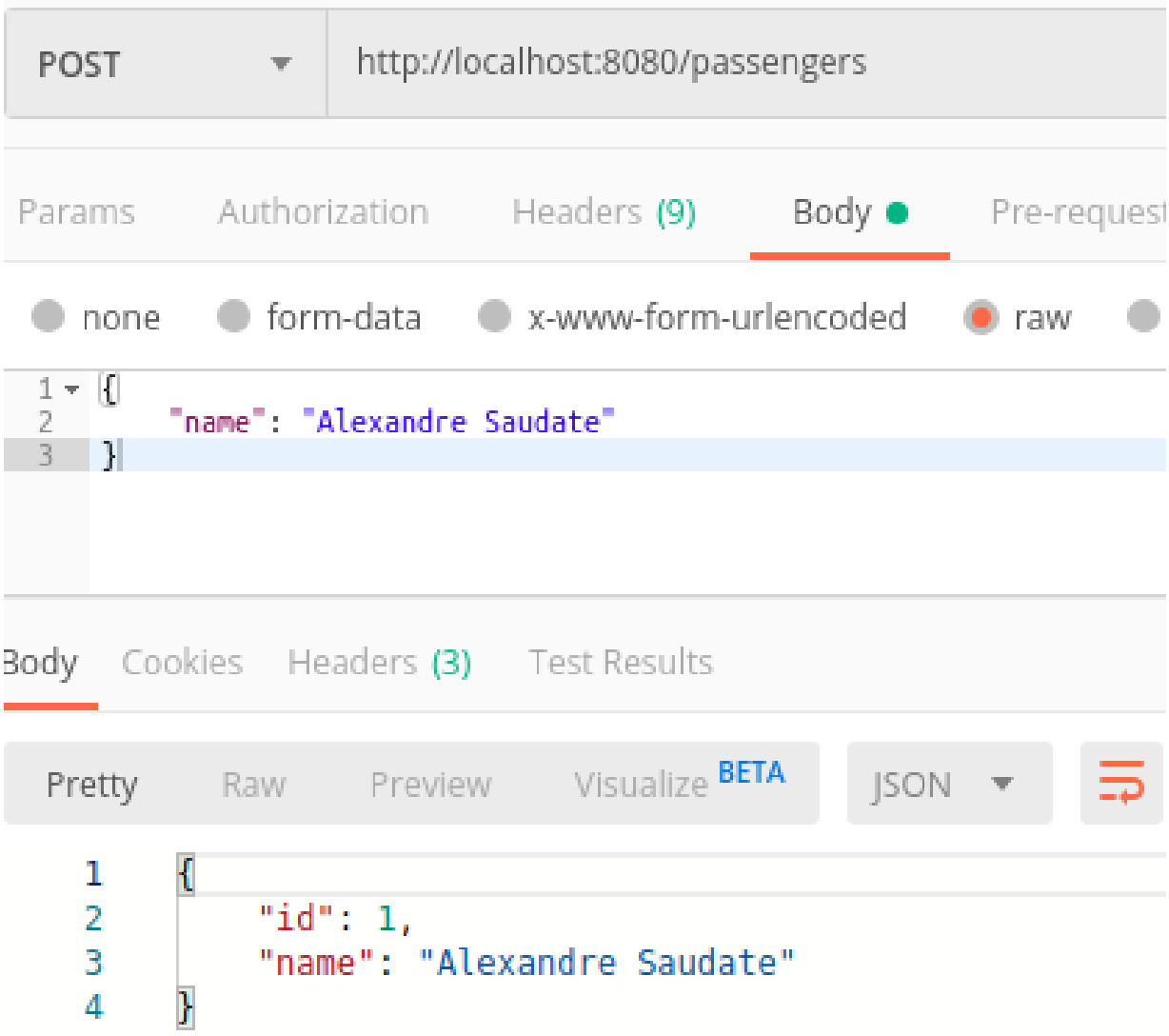


Figura 3.1: Realizando a criação do passageiro

Na sequência, vamos utilizar o `id` do passageiro para enviar a requisição de solicitação de viagem:

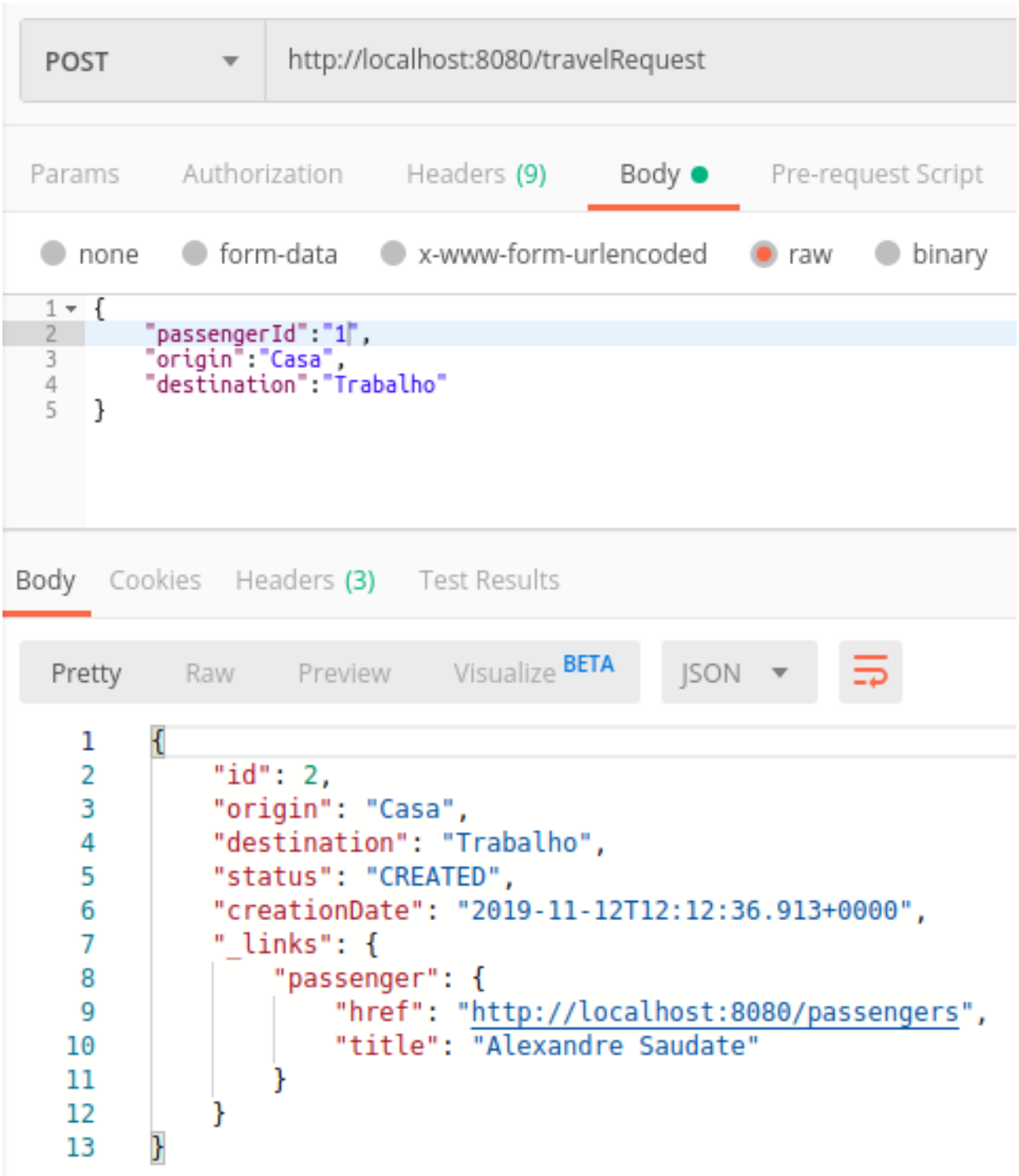


Figura 3.2: Realizando a criação da solicitação de viagem

Note a presença do JSON contendo o link para o passageiro:

```
{
    "id": 2,
```

```
    "origin": "Casa",
    "destination": "Trabalho",
    "status": "CREATED",
    "creationDate": "2019-11-12T12:12:36.913+0000",
    "_links": {
        "passenger": {
            "href": "http://localhost:8080/passengers",
            "title": "Alexandre Saudate"
        }
    }
}
```

No entanto, observe que o atributo `href` está incompleto. Para referenciar o passageiro exato, o ID do passageiro deveria estar na URL. Por que isso aconteceu?

Quando utilizamos o método `linkTo` da classe `WebMvcLinkBuilder`, passamos como parâmetro a classe `PassengerAPI`. Isso dá ao `builder` a capacidade de encontrar a API mais geral de passageiros (como de fato aconteceu), mas não a API específica para listar um único passageiro. Nesse caso, temos duas alternativas: construir manualmente o restante da URL ou fornecer para o `builder` o método específico para listar um único passageiro. Vamos seguir pela construção manual. Para isso, vamos utilizar o método `slash` do `builder`, passando como parâmetro o ID do passageiro:

```
val passengerLink = WebMvcLinkBuilder
    .linkTo(PassengerAPI::class)
    .slash(travelRequest.passenger.id)
    .withRel("passenger")
    .withTitle(travelRequest.passenger.name)
```

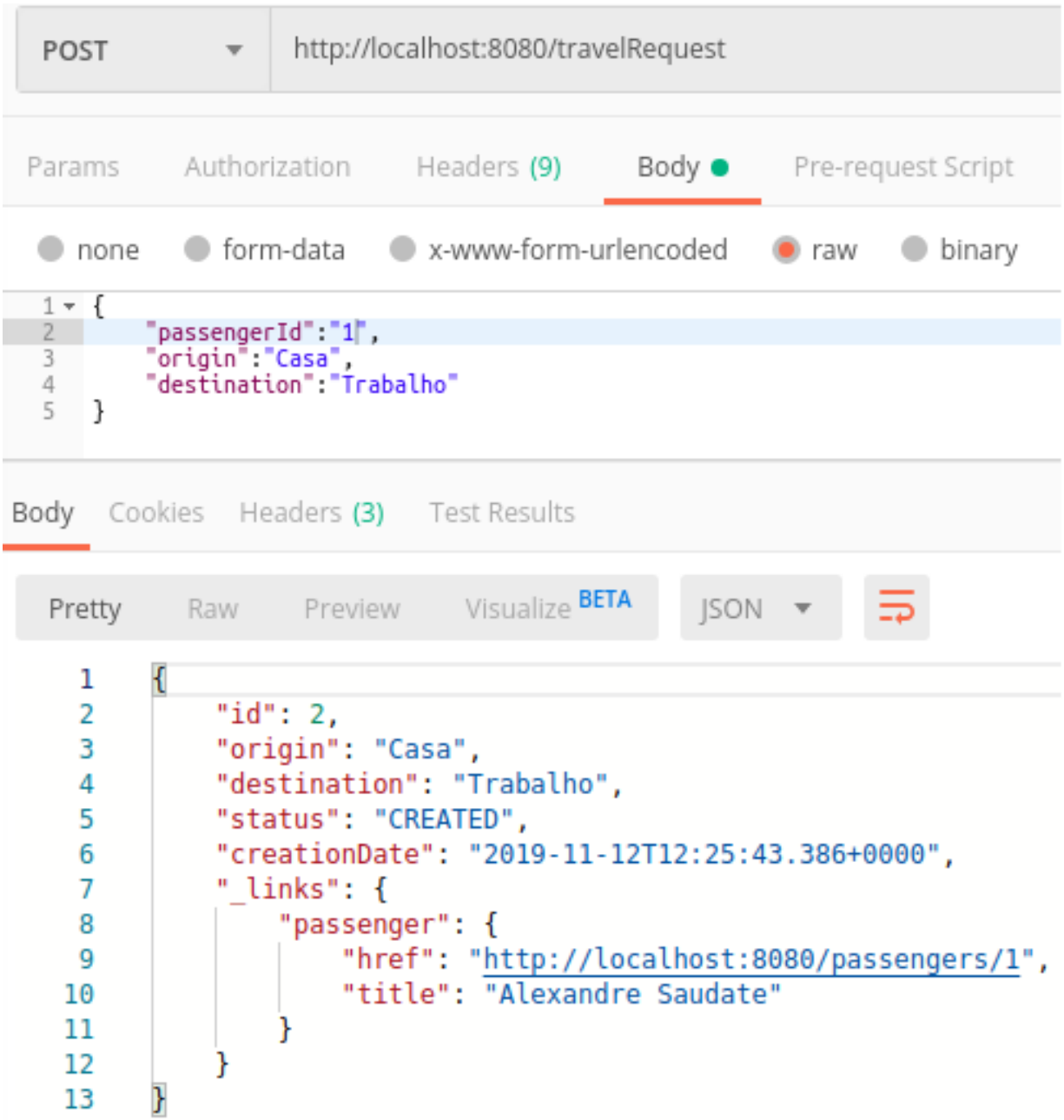


Figura 3.3: Refazendo a solicitação de viagem

Observe que o link está na forma correta agora:

```
{
    "id": 2,
    "origin": "Casa",
    "destination": "Trabalho",
    "status": "CREATED",
```

```
    "creationDate": "2019-11-12T12:25:43.386+0000",
    "_links": {
        "passenger": {
            "href": "http://localhost:8080/passengers/1",
            "title": "Alexandre Saudate"
        }
    }
}
```

Conclusão

Neste capítulo, vimos um pouco sobre como recursos referenciam uns aos outros até conhecermos a técnica HATEOAS. Veremos agora como um motorista vai aceitar corridas próximas a ele - o que vai nos levar a construir um cliente REST para a API do Google Maps.

Capítulo 4
Criando clientes REST

"O sucesso depende de preparação prévia." - Confúcio.

Até aqui, vimos o básico sobre algumas técnicas de REST no lado do servidor. Para isso, construímos uma API minimamente funcional na qual as solicitações de viagens são criadas. Mas e quanto à API de aceite de solicitações de viagens? Como fica?

Neste capítulo, vamos criar essa API e ver como construir um cliente para a API do Google Maps para que o motorista receba somente as solicitações de viagens de pessoas que estiverem próximas a ele.

4.1 Reorganizando o projeto

Em primeiro lugar, vamos acomodar as classes que vão receber a estrutura. Nosso projeto está seguindo a seguinte arquitetura:

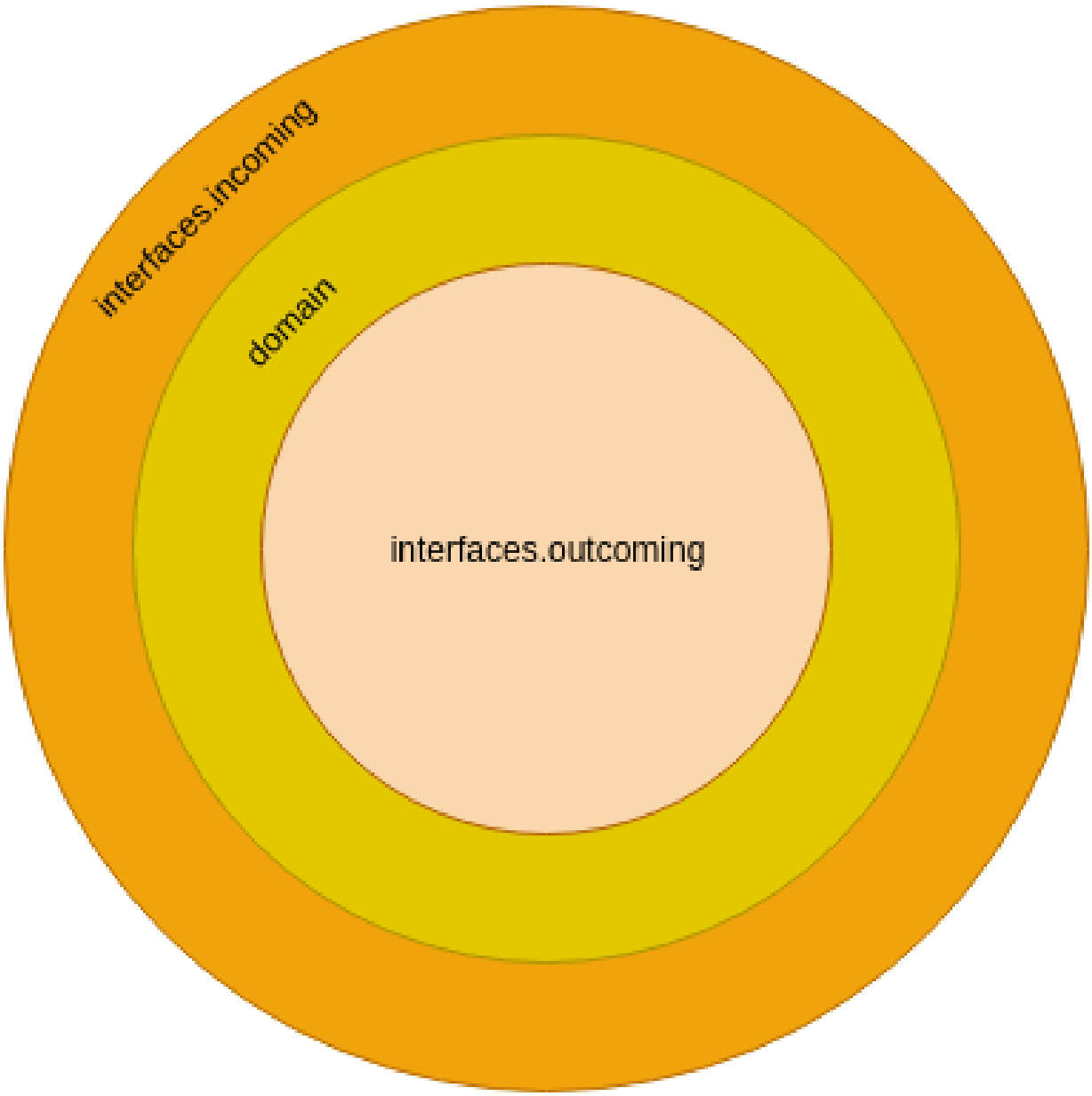


Figura 4.1: Diagrama de arquitetura do projeto

Procurei unificar nessa arquitetura o conceito de *clean architecture* segundo Uncle Bob (MARTIN, 2012) com alguns conceitos de *Domain-Driven Design*, conforme descrito por Eric Evans (EVANS, 2003). Assim sendo, note que é uma arquitetura que mimetiza uma cebola: as requisições entram pela camada mais externa (as APIs que estão no pacote `interfaces.incoming`) e vão penetrando até

chegar às interfaces de saída que estão presentes no pacote `interfaces.outcoming`.

No nosso projeto, as interfaces de entrada são APIs REST, mas poderiam também ser *listeners* de mensagens JMS, por exemplo. A parte de domínio contém a nossa lógica de negócio, que no caso é representada pelas entidades e repositórios (lembrando que, de acordo com o DDD, repositórios fazem parte do domínio). Finalmente, as interfaces de saída são clientes de outras aplicações. No nosso caso, é onde estará localizado o cliente para o Google Maps.

Sendo assim, rearranjamos os pacotes da aplicação para acomodar essa estrutura:

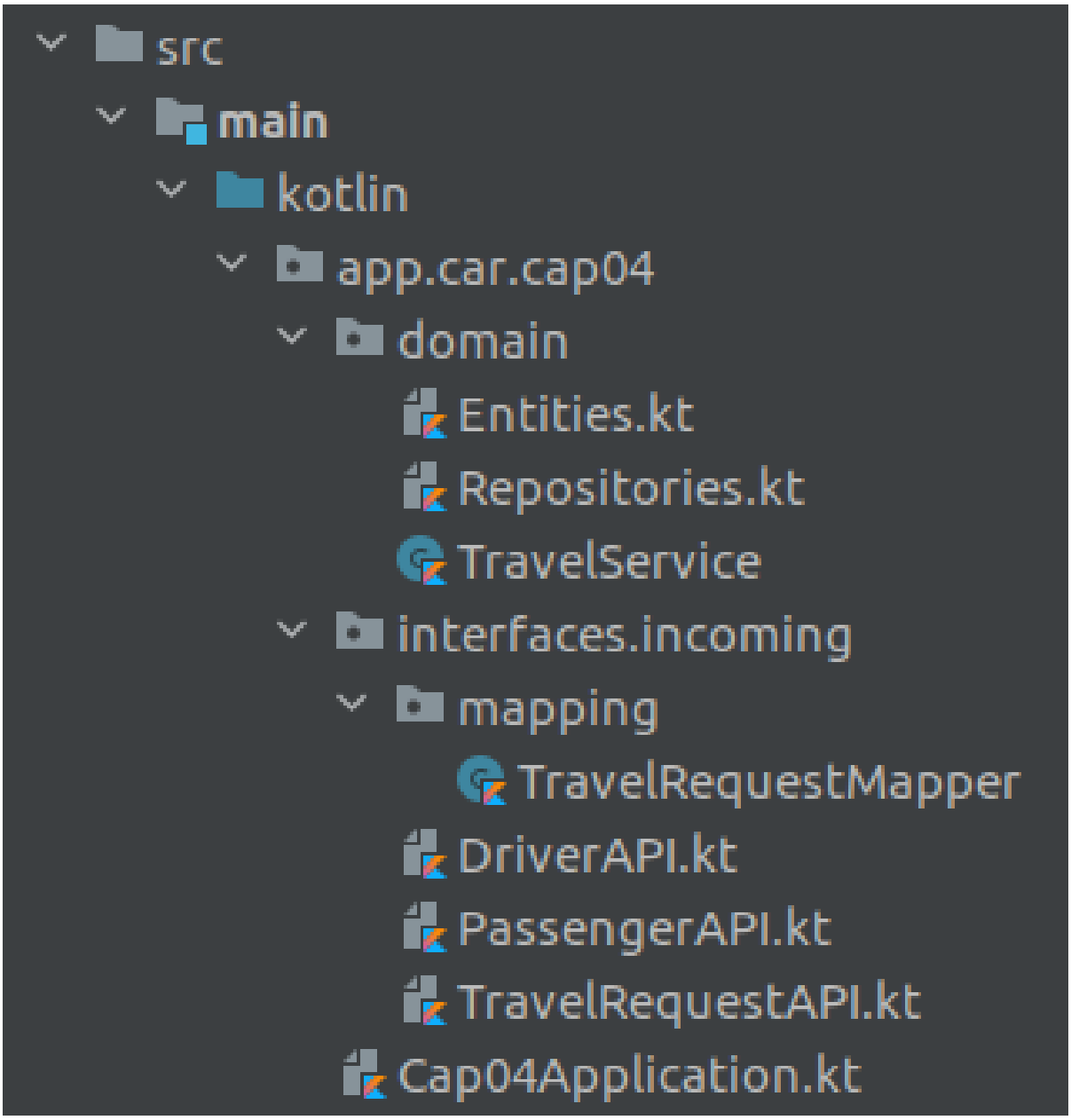


Figura 4.2: Reorganização dos pacotes

4.2 Criando a chave de API do Google

Vamos conhecer a página do Google. O serviço que queremos é o que fornece rotas entre localizações. Tudo isso está descrito em

https://developers.google.com/maps/documentation/directions/start#sample-request.

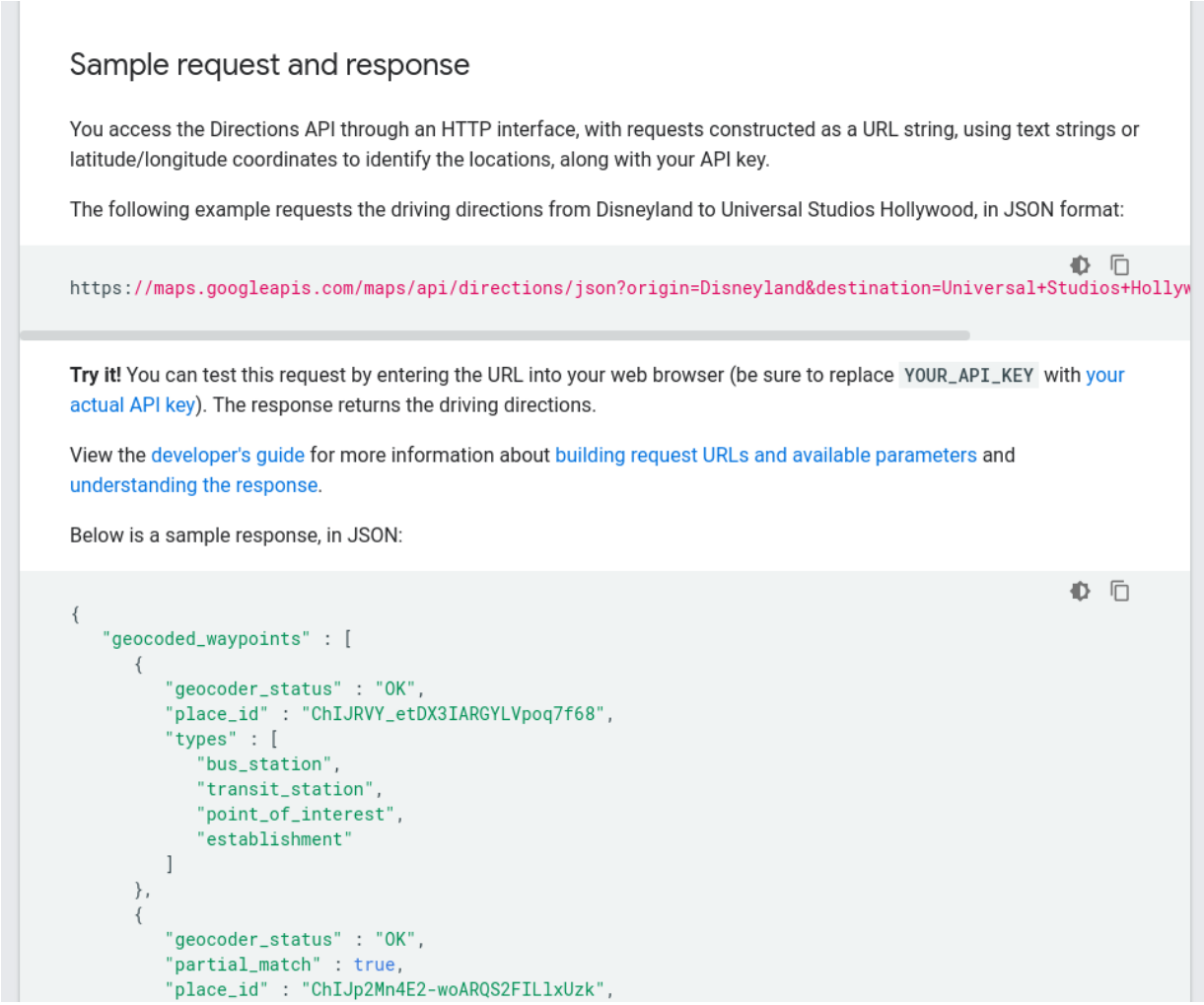

Sample request and response
You access the Directions API through an HTTP interface, with requests constructed as a URL string, using text strings or latitude/longitude coordinates to identify the locations, along with your API key.
The following example requests the driving directions from Disneyland to Universal Studios Hollywood, in JSON format:
https://maps.googleapis.com/maps/api/directions/json?origin=Disneyland&destination=Universal+Studios+Hollyw
Try it! You can test this request by entering the URL into your web browser (be sure to replace YOUR_API_KEY with your actual API key). The response returns the driving directions.
View the developer's guide for more information about building request URLs and available parameters and understanding the response.
Below is a sample response, in JSON:
{
"geocoded_waypoints" : [
{
"geocoder_status" : "OK",
"place_id" : "ChIJRVY_etDX3IARGYLVpoq7f68",
"types" : [
"bus_station",
"transit_station",
"point_of_interest",
"establishment"
]
},
{
"geocoder_status" : "OK",
"partial_match" : true,
"place_id" : "ChIJp2Mn4E2-woARQS2FILlxUzk",

Figura 4.3: Apresentação da página da API do Google Maps

Observe que a página traz um link de exemplo: https://maps.googleapis.com/maps/api/directions/json?origin=Disneyland&destination=Universal+Studios+Hollywood&key=YOUR_API_KEY. Ao copiar e colar essa URL em nosso *browser*, temos a seguinte resposta:

```
{
"error_message": "The provided API key is invalid.",
"routes": [],
```

```
"status": "REQUEST_DENIED"
}
```

Isso quer dizer que a chave de API é inválida. Mas vamos analisar a URL fornecida: a seção `https://maps.googleapis.com/maps/api/directions/json` é a base da API. A partir do sinal de interrogação, temos os chamados `query parameters`, ou seja, o sistema de parametrização da API. Eles são pares de chave-valor separados por `&`, então temos os parâmetros `origin`, `destination` e `key`. Os valores são `Disneyland`, `Universal Studios Hollywood` e `YOUR_API_KEY`, respectivamente. Perceba que, de fato, a chave da API é apenas um *placeholder*.

Para conseguir uma chave de API, existe uma seção na própria página demonstrando isso:

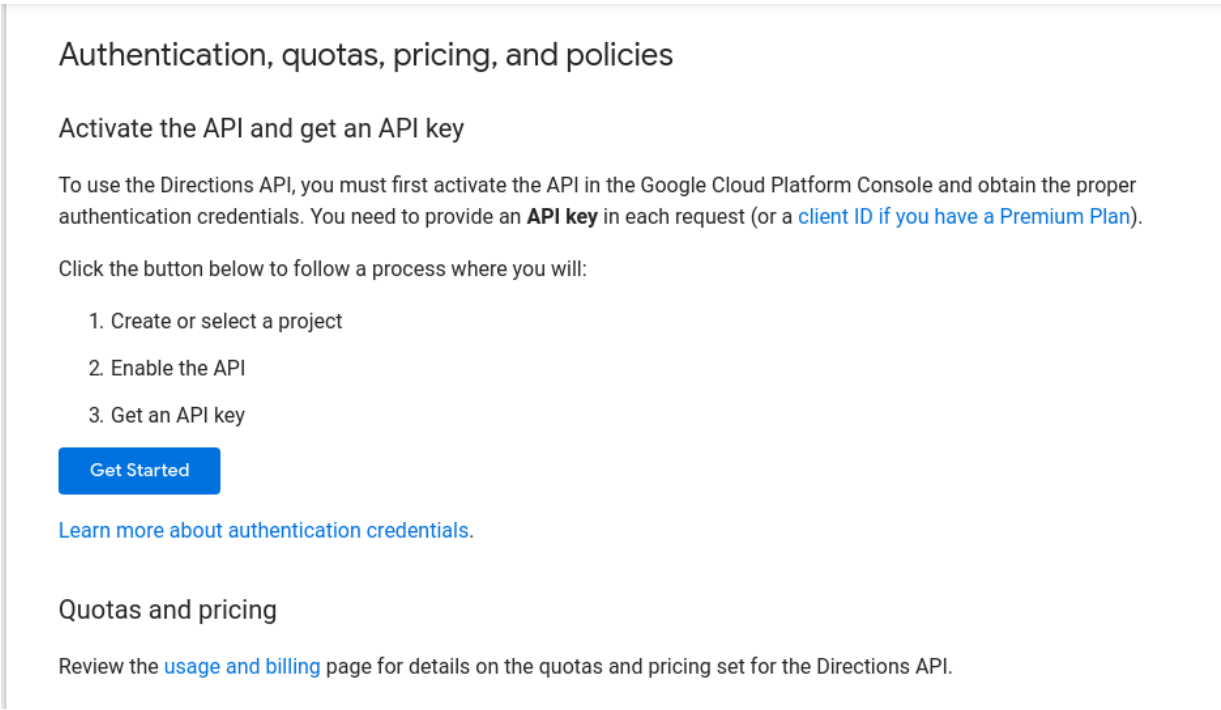

Authentication, quotas, pricing, and policies
Activate the API and get an API key
To use the Directions API, you must first activate the API in the Google Cloud Platform Console and obtain the proper authentication credentials. You need to provide an API key in each request (or a client ID if you have a Premium Plan).
Click the button below to follow a process where you will:
1. Create or select a project
2. Enable the API
3. Get an API key
Get Started
Learn more about authentication credentials.
Quotas and pricing
Review the usage and billing page for details on the quotas and pricing set for the Directions API.

Figura 4.4: Ativação da API do Google

As instruções dizem para criar um projeto, habilitar a API e então conseguir uma chave de API. Vamos seguir o procedimento, clicando no botão `Get Started`. Ao fazer isso, somos levados a uma

outra página, que nos pergunta quais APIs queremos utilizar, dando as opções do Google Maps, Routes e Places. Vamos marcar os dois primeiros e seguir:

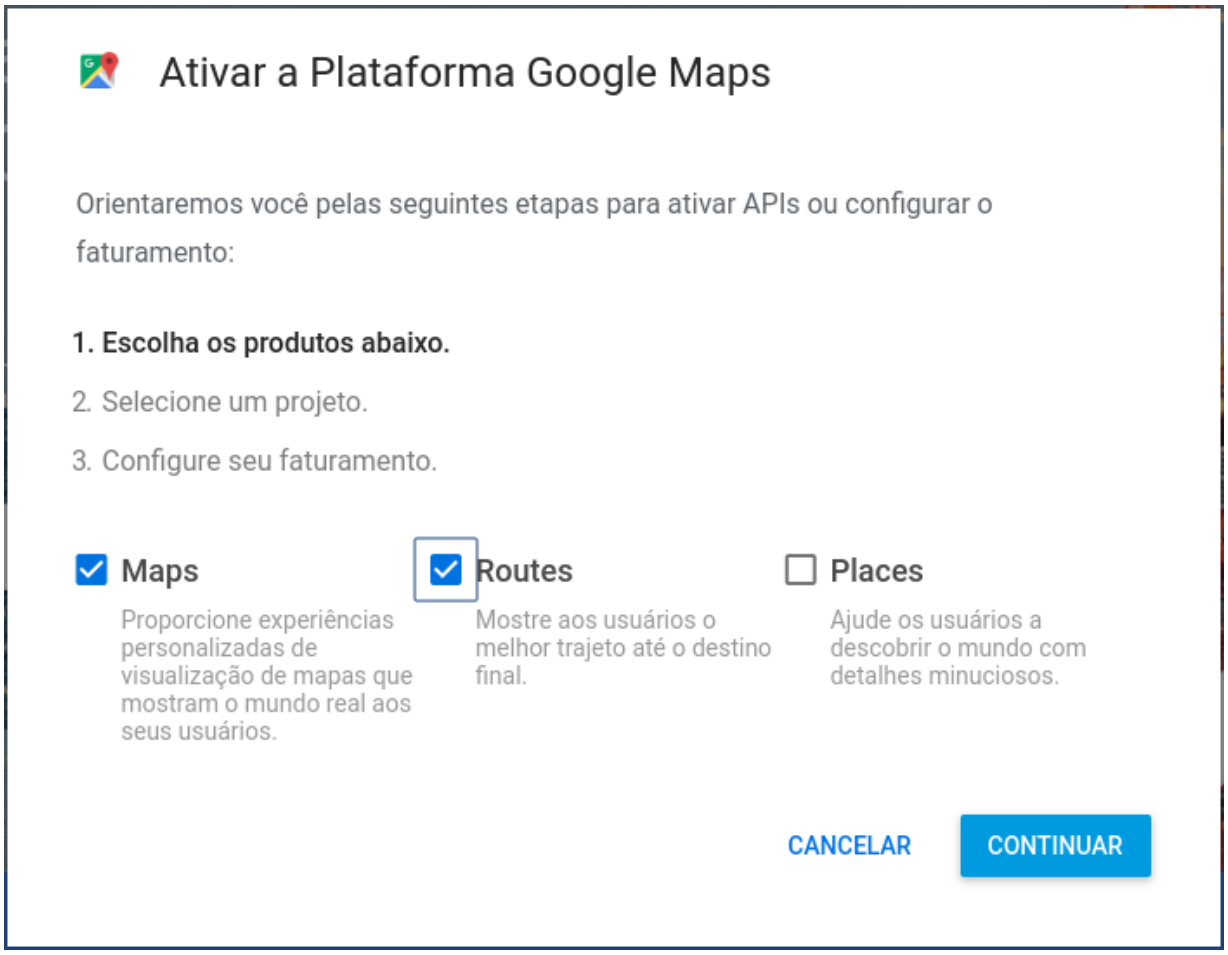

Ativar a Plataforma Google Maps
Orientaremos você pelas seguintes etapas para ativar APIs ou configurar o faturamento:
1. Escolha os produtos abaixo.
2. Selecione um projeto.
3. Configure seu faturamento.
Maps
Proporcione experiências personalizadas de visualização de mapas que mostram o mundo real aos seus usuários.
Routes
Mostre aos usuários o melhor trajeto até o destino final.
Places
Ajude os usuários a descobrir o mundo com detalhes minuciosos.
CANCELAR
CONTINUAR

Figura 4.5: Ativação do Google Maps

A seguir, ele nos pergunta qual é o projeto. Vou criar o projeto `RESTBook` e continuar (caso você siga esse passo a passo, pode colocar o nome que preferir, pois isso não afeta a funcionalidade):

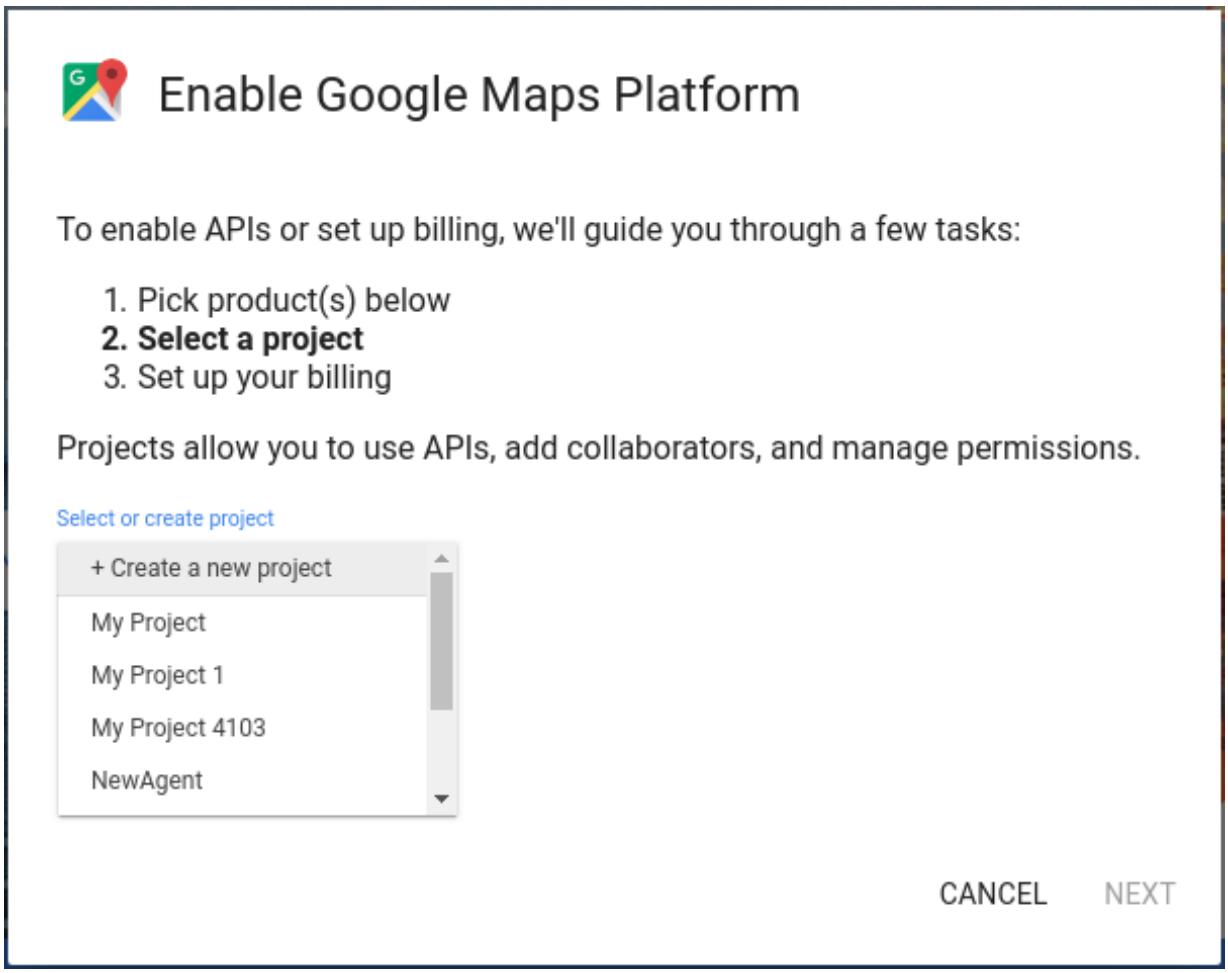

Enable Google Maps Platform
To enable APIs or set up billing, we'll guide you through a few tasks:
1. Pick product(s) below
2. Select a project
3. Set up your billing
Projects allow you to use APIs, add collaborators, and manage permissions.
Select or create project
+ Create a new project
My Project
My Project 1
My Project 4103
NewAgent
CANCEL
NEXT

Figura 4.6: Criação do projeto

Se ainda não tiver uma conta do Google, você será direcionado para a tela de ativação da cobrança:

Ativar o faturamento do projeto "RESTBook"
Você não é administrador de nenhuma conta de faturamento. Para ativar o faturamento neste projeto, crie uma nova conta de faturamento ou entre em contato com o administrador da conta de faturamento para que ele o ative para você. Saiba mais
CANCELAR
CRIAR CONTA DE FATURAMENTO

Figura 4.7: Ativação do billing

Não se preocupe com essa cobrança. Conforme mostrado na tela seguinte, o Google oferece (até o momento da escrita deste livro) 300 dólares para serem gastos nos próximos 12 meses. Ou seja, você pode usar livremente os recursos da plataforma até um limite de 300 dólares (os valores cobrados costumam girar em torno de alguns centavos). Conforme diz a página, o cartão de crédito é requerido para fins de validação, mas nenhuma cobrança será feita sem seu consentimento:

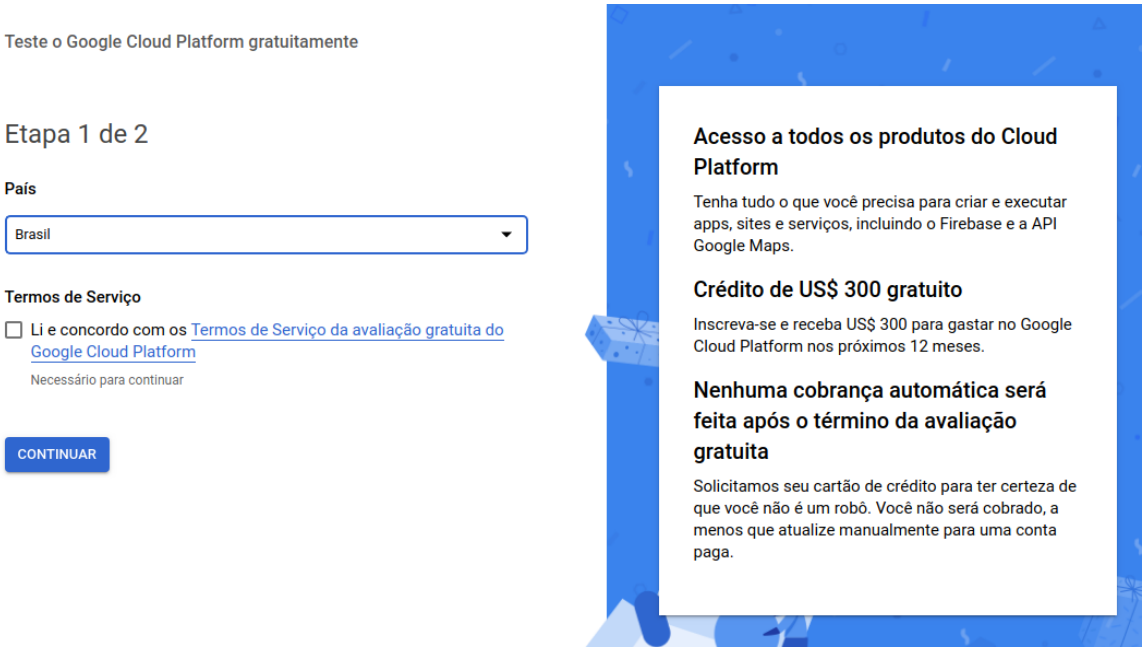

Teste o Google Cloud Platform gratuitamente
Etapa 1 de 2
País
Brasil
Termos de Serviço
Li e concordo com os Termos de Serviço da avaliação gratuita do Google Cloud Platform
Necessário para continuar
CONTINUAR
Acesso a todos os produtos do Cloud Platform
Tenha tudo o que você precisa para criar e executar apps, sites e serviços, incluindo o Firebase e a API Google Maps.
Crédito de US$ 300 gratuito
Inscreva-se e receba US$ 300 para gastar no Google Cloud Platform nos próximos 12 meses.
Nenhuma cobrança automática será feita após o término da avaliação gratuita
Solicitamos seu cartão de crédito para ter certeza de que você não é um robô. Você não será cobrado, a menos que atualize manualmente para uma conta paga.

Figura 4.8: Primeira etapa da criação do billing

No decorrer do processo, haverá um formulário a ser preenchido com o endereço e o cartão de crédito. Preencha os dados e clique em `Iniciar minha avaliação gratuita`:

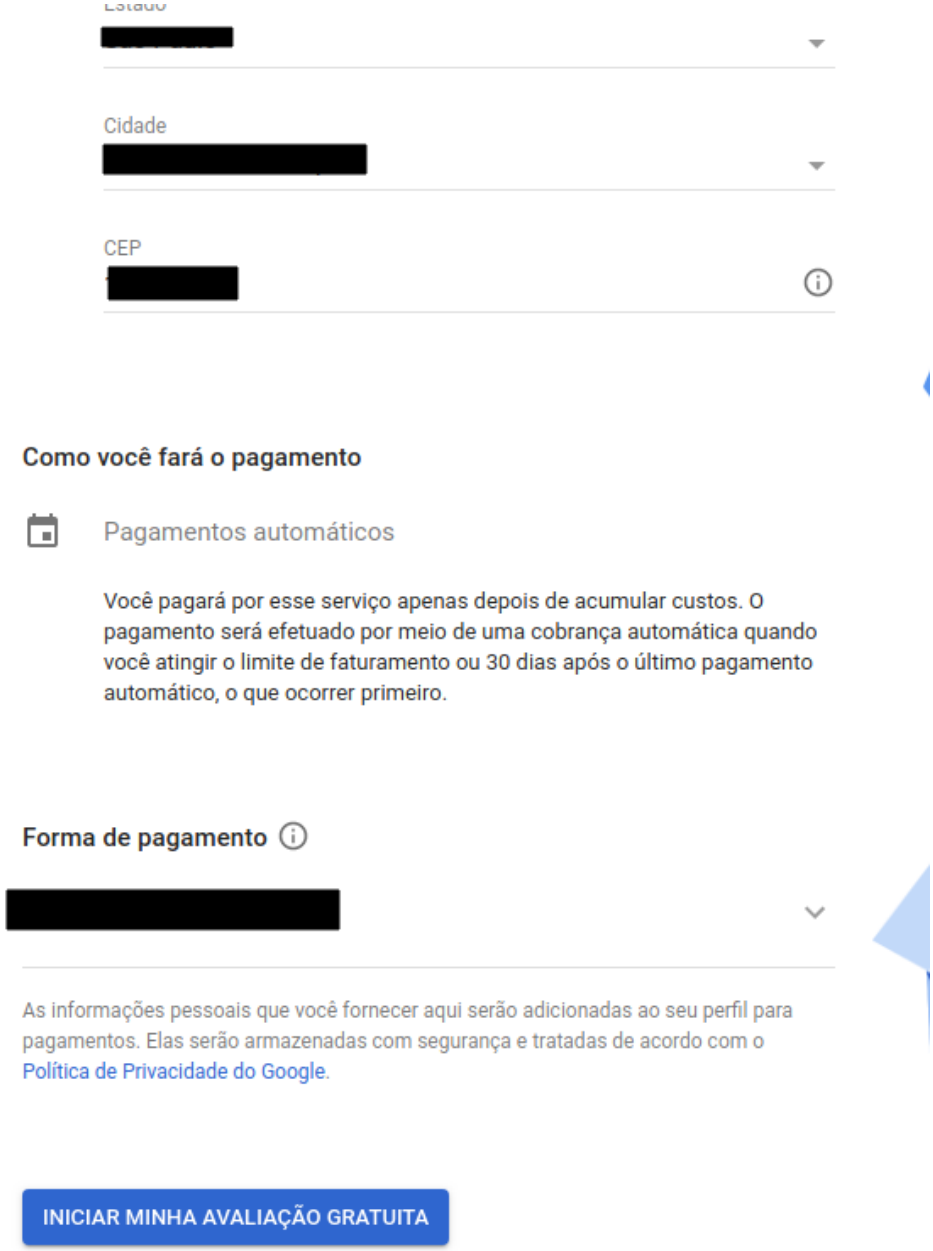

Cidade
CEP
Como você fará o pagamento
Pagamentos automáticos
Você pagará por esse serviço apenas depois de acumular custos. O pagamento será efetuado por meio de uma cobrança automática quando você atingir o limite de faturamento ou 30 dias após o último pagamento automático, o que ocorrer primeiro.
Forma de pagamento
As informações pessoais que você fornecer aqui serão adicionadas ao seu perfil para pagamentos. Elas serão armazenadas com segurança e tratadas de acordo com o Política de Privacidade do Google.
INICIAR MINHA AVALIAÇÃO GRATUITA

Figura 4.9: Ativando billing

Ao concluir o processo, um último *pop-up* vai ser aberto para reforçar essas informações:

Google Cloud Platform
Olá, Alexandre,
Agradecemos por você se inscrever na avaliação gratuita de 12 meses.
Agradecemos sua inscrição. A avaliação gratuita inclui US$ 300 em crédito para gastar nos próximos 12 meses. Se os créditos acabarem, não se preocupe. Você não será cobrado sem sua permissão.
OK

Figura 4.10: Confirmação de ativação do billing Google Cloud

Depois dessas etapas, você será levado ao painel de gerenciamento do Google Cloud, no qual haverá um *pop-up* semelhante ao seguinte:

Ativar o Google Maps Platform
Ativar as APIs
Isso ativará 10 API(s) Google Maps Platform e criará uma chave de API para a implementação.
CANCELAR
PRÓXIMA

Figura 4.11: Pop-up de ativação do Google Maps Platform

Ao clicar em `Próxima`, uma tela semelhante à seguinte deverá surgir:

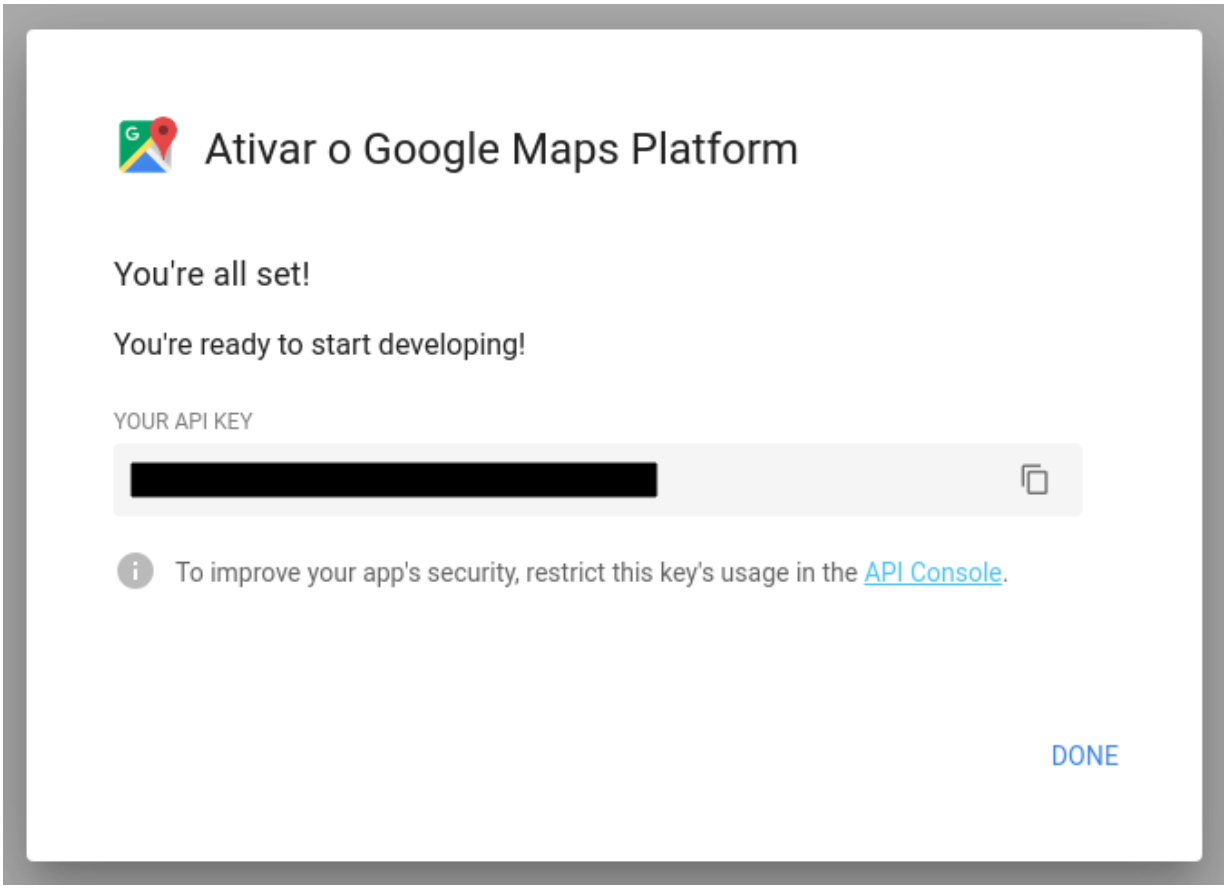

Ativar o Google Maps Platform
You're all set!
You're ready to start developing!
YOUR API KEY
[redacted]
To improve your app's security, restrict this key's usage in the API Console.
DONE

Figura 4.12: Pop-up com a API Key

Copie a `API Key`. Vamos refazer a requisição que fizemos no início do capítulo para o Google Maps, mas, desta vez, vamos substituir o `YOUR_API_KEY` pela `API Key` que você acabou de conseguir. O resultado é um JSON contendo o descritivo do Google Maps, como o seguinte:

```
{
   "geocoded_waypoints": [
      {
         "geocoder_status": "OK",
         "place_id": "ChIJ96XKNOZ-3YgRoPEc0B85Hqc",
         "types": [
            "amusement_park",
            "establishment",
            "point_of_interest",
            "tourist_attraction",
            "travel_agency"
         ]
      },
      {
         "geocoder_status": "OK",
         "place_id": "ChIJzzgyJU--woARcZqceSdQ3dM",
         "types": [
            "amusement_park",
            "establishment",
            "point_of_interest",
            "tourist_attraction"
         ]
      }
   ],
   "routes": [
      {
         "bounds": {
            "northeast": {
               "lat": 34.1358593,
               "lng": -81.5573272
            },
            "southwest": {
               "lat": 28.3780416,
               "lng": -118.3511633
            }
         },
         "copyrights": "Dados cartográficos ©2019 Google, INEGI",
         "legs": [
            {
               "distance": {
                  "text": "2.520 mi",
                  "value": 4056271
               },
               "duration": {
                  "text": "1 dia 12 horas",
                  "value": 129180
               },
               "end_address": "100 Universal City Plaza, Universal City, CA 91608, EUA",
               "end_location": {
                  "lat": 34.1358593,
                  "lng": -118.3511633
               },
               "start_address": "Walt Disney World Resort, Orlando, FL 32830, EUA",
               "start_location": {
                  "lat": 28.3780416,
                  "lng": -81.6037027
               },
               "steps": [
                  {
```

Figura 4.13: Resposta do Google Maps

4.3 Criando o código do cliente

Agora que já temos a comunicação com o serviço do Google Maps estabelecida, vamos criar a classe para realizar a busca do tempo de distância automaticamente. Primeiramente, vamos criar uma classe com um método `getDistanceBetweenAddresses`, cujo retorno seja um Integer:

```
package app.car.cap04.interfaces.outcoming

import org.springframework.stereotype.Service

@Service
class GMapsService {

    fun getDistanceBetweenAddresses(addressOne: String, addressTwo:
String) : Int {
        return 0
    }
}
```

O próximo passo é extrair o endereço que vimos anteriormente para um *template*. Isso significa que vamos retirar os valores da *query string* e substituí-los por pares de template (que são delimitados por chaves `{}`). Vamos então colocar esse endereço em uma constante:

```
val GMAPS_TEMPLATE: String =
    "https://maps.googleapis.com/maps/api/directions/json?origin=
{origin}&destination={destination}&key={key}"
```

Feito isso, vamos utilizar um recurso do Spring que injeta alguns valores - no nosso caso, é interessante retirar a `API KEY` do projeto e colocar em um arquivo chamado `application.properties`, que fica dentro da pasta do nosso projeto `src/main/resources`:

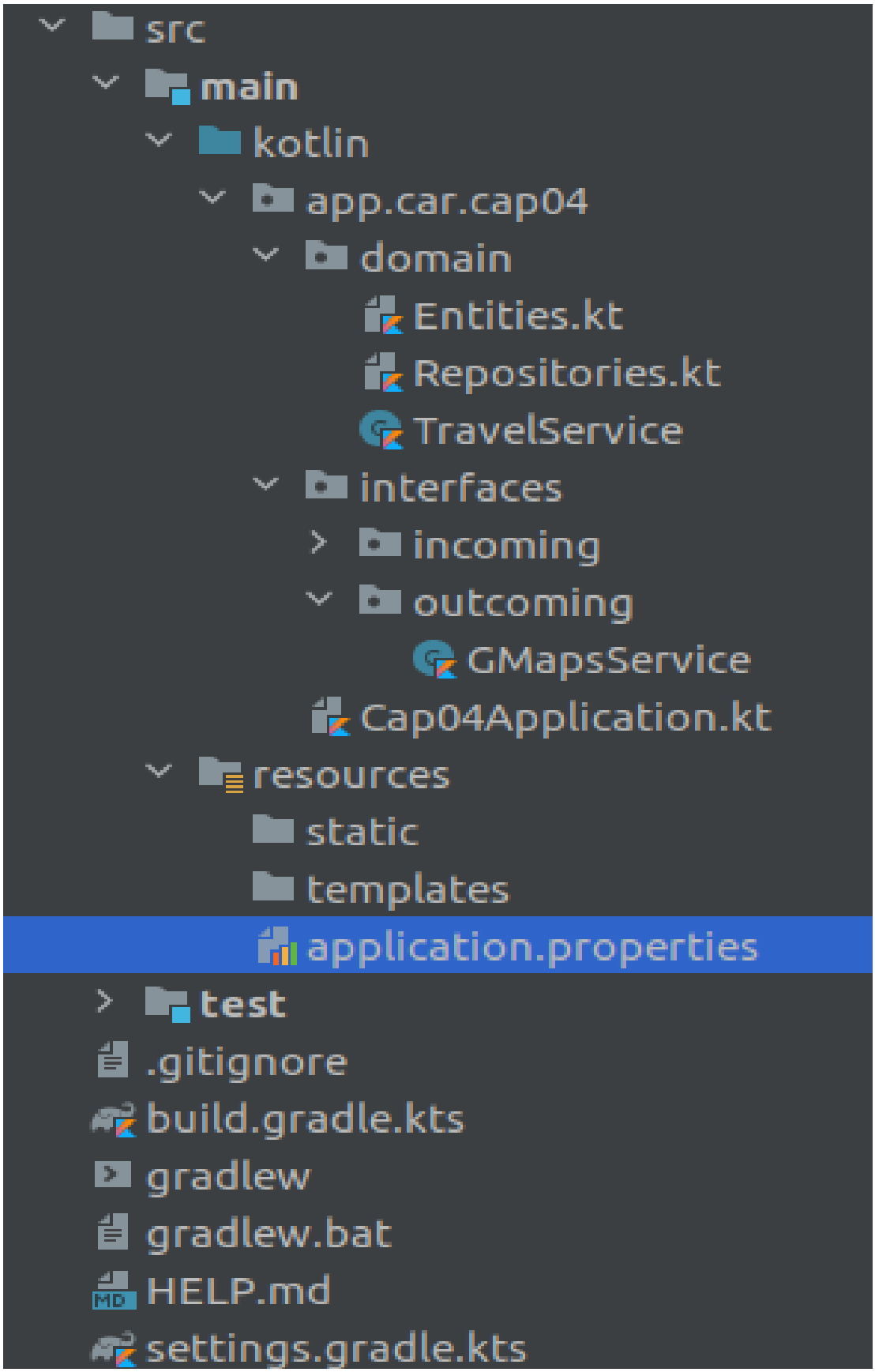

src
main
kotlin
app.car.cap04
domain
Entities.kt
Repositories.kt
TravelService
interfaces
incoming
outcoming
GMapsService
Cap04Application.kt
resources
static
templates
application.properties
test
.gitignore
build.gradle.kts
gradlew
gradlew.bat
HELP.md
settings.gradle.kts

Figura 4.14: Localização do arquivo application.properties

Dentro desse arquivo, vamos colocar uma propriedade com a `API KEY` do Google Maps:

```
app.car.domain.googlemaps.apikey=SUA API KEY
```

> **ONDE COLOCAR A API KEY?**
>
> Em uma aplicação no mundo real, é preferível que essa chave não fique no arquivo de propriedades, pois pode ser vazada por desenvolvedores mal-intencionados. Hoje em dia, contamos com guias, como as aplicações de 12 fatores (WIGGINS, 2017) para nos orientar a respeito de como lidar com essas propriedades. O terceiro fator trata sobre configurações desse tipo, sendo que o preferível é armazená-las em variáveis de ambiente. Dessa forma, apenas as pessoas responsáveis por realizar a manutenção da aplicação em produção poderiam ter acesso ao valor real da chave, limitando o acesso a ela.
>
> Além de variáveis de ambiente, existem outras variantes caso você use um *cloud provider* como a AWS. A AWS conta com um serviço específico chamado `AWS Systems Manager` (SSM), que trata de maneira especializada o acesso a parâmetros sensíveis como esse. Usando o SSM, é possível garantir que apenas a aplicação e a pessoa que gerou a *API Key* tenham acesso a ela.

De volta à classe `GMapsService`, vamos criar um atributo para receber esse valor, através do uso da anotação `@Value` do Spring. Essa anotação recebe como parâmetro uma expressão que vai extrair o valor da propriedade e injetá-la nesse atributo:

```
package app.car.cap04.interfaces.outcoming

import org.springframework.beans.factory.annotation.Value
import org.springframework.stereotype.Service

@Service
```

```
class GMapsService(
    @Value("\${app.car.domain.googlemaps.apikey}")
    val appKey: String
) {

    val GMAPS_TEMPLATE: String =
"https://maps.googleapis.com/maps/api/directions/json?origin=
{origin}&destination={destination}&key={key}"

    fun getDistanceBetweenAddresses(addressOne: String, addressTwo:
String) : Int {
        return 0
    }
}
```

POR QUE A EXPRESSÃO EM KOTLIN PRECISA TER UMA BARRA INVERTIDA ANTES DO $?

O Kotlin implementa uma capacidade de algumas linguagens conhecida como interpolação de strings, ou seja, uma string pode detectar o valor de uma variável e automaticamente inserir o valor dessa variável dentro da String. Essa capacidade, no entanto, não é automática - é sinalizada a partir do uso do cifrão (`$`). O cifrão é um caractere reservado dentro de Strings em Kotlin, por isso precisamos colocar a barra invertida na frente, para que o Kotlin não trate o cifrão como o início de uma interpolação de strings.

Agora, vamos passar à codificação do cliente, efetivamente. Vamos usar como apoio uma classe do Spring chamada `RestTemplate` . Essa classe fornece métodos para consumirmos serviços REST de diversas formas e vai nos ajudar a consumir o serviço do Google. Antes, no entanto, precisamos analisar o serviço e de que forma precisamos do resultado dele.

Uma análise da árvore de dados nos mostra que a informação de que precisamos está no elemento `legs` , onde existe uma porção

chamada `duration` com um campo `value` (no nosso exemplo, esse dado tem o valor de `129180`). Temos várias opções para extrair essa informação: poderíamos criar a estrutura de objetos em Kotlin que representariam a árvore inteira. Porém, isso nos consumiria algum tempo para realizar o mapeamento e seria propenso a não se manter estável no caso de haver mudanças na API do Google.

Nossa segunda opção seria realizar a extração para o formato de mapas em Kotlin. Apesar de essa estrutura ser um pouco mais flexível, ainda seria trabalhoso realizar a busca do dado específico (pois o Kotlin mapearia para mapas onde os valores seriam outros mapas, ou seja, haveria uma complexidade para a busca do valor). Além disso, essa extração também seria propensa a erros, pois caso houvesse qualquer mudança nas estruturas que fazem o acesso ao nosso dado específico, não seríamos mais capazes de localizar a informação de que precisamos.

Nossa terceira opção seria utilizar uma linguagem de busca específica, chamada `JSONPath`. Essa linguagem foi criada com o propósito específico de localizar informações dentro de árvores JSON, e uma de suas capacidades é justamente localizar dados com o fornecimento de apenas parte das informações necessárias.

Seguindo com o desenvolvimento do nosso serviço, então, primeiramente vamos instanciar o nosso `RestTemplate`. Existem seis opções de consulta utilizando o método `GET`, que se subdividem em dois tipos principais, `getForEntity` e `getForObject`. Essas opções são dadas para que o usuário decida se quer que o método retorne um `ResponseEntity` ou o resultado diretamente. A abstração fornecida por essa classe nos dá a habilidade de recuperar os cabeçalhos, códigos de status etc., da requisição. Se esses dados não nos interessam (o que é o caso no momento), então podemos utilizar `getForObject` diretamente.

O método `getForObject` é sobrecarregado por conta das formas de se passar os valores dos *placeholders* na URL. Note que, como a URL do Google Maps está em formato de template, temos que

fornecer os dados de alguma forma. Pois o método é sobrecarregado para que passemos esses dados em formato de `varargs`, em formato de mapa, ou simplesmente não passemos nada. Vamos utilizar o formato de `varargs` e fornecer os dados na sequência em que aparecem no *template*, isto é, o endereço de origem, de destino e a chave:

```
val template = RestTemplate()
val jsonResult = template.getForObject(GMAPS_TEMPLATE, String::class.java,
addressOne, addressTwo, appKey)
```

4.4 Recuperando os dados com JSONPath

O próximo passo é utilizar o `JSONPath` para realizar essa busca. Antes, no entanto, precisamos adicionar a dependência de uma biblioteca capaz de fornecer essa capacidade:

```
implementation("com.jayway.jsonpath:json-path")
```

A linguagem `JSONPath` funciona da seguinte forma: primeiramente, as buscas começam com um `$` (tomando o cuidado de realizar o escape, conforme mencionado anteriormente). Depois, colocamos um `.` e o nome do elemento que queremos consultar. Caso queiramos vários elementos, vamos separando cada um com `.`. O site https://jsonpath.com/ ilustra isso:

JSONPath Online Evaluator - jsonpath.com

Inputs

Output paths

JSONPath Syntax

$.phoneNumbers[:1].type

Example '$.phoneNumbers[*].type' See also JSONPath expressions

JSON

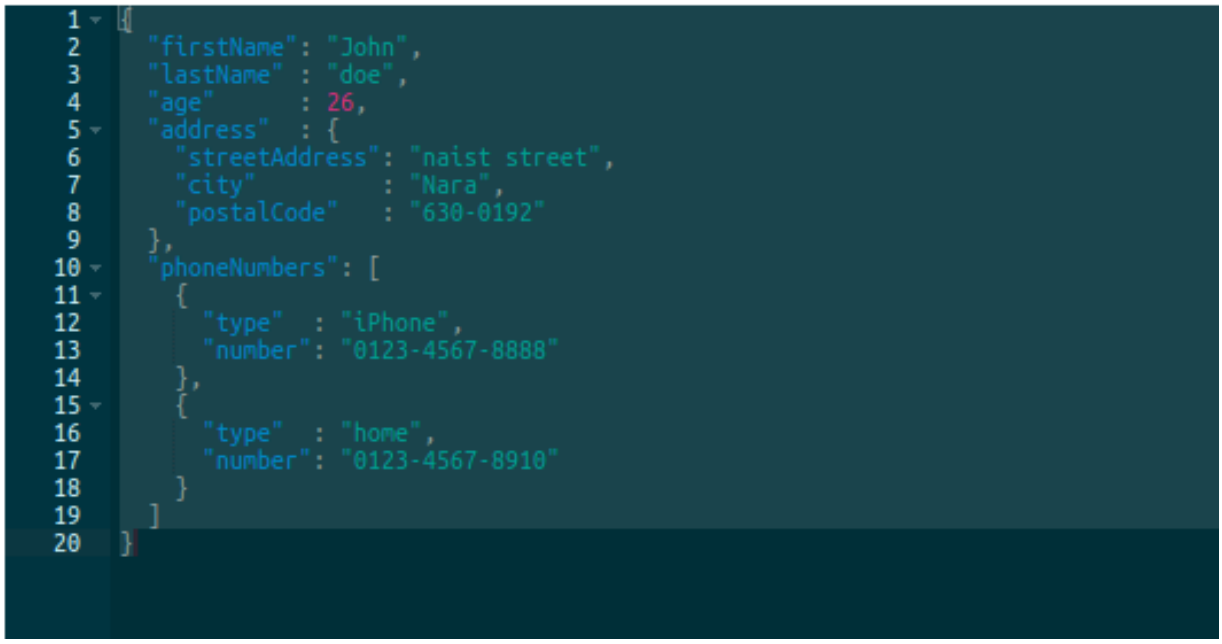


Figura 4.15: Exemplo do JSONPath.com de entrada

Evaluation Results

Figura 4.16: Exemplo do JSONPath.com de saída

Observe que podemos usar esse site para elaborar o resultado que queremos. Copie o conteúdo do JSON do Google Maps para o campo `Inputs` do site para que possamos começar as nossas verificações.

Segundo o que observamos, note que o elemento `routes` está à frente do elemento `legs`. Não nos interessa a declaração do elemento - portanto, podemos utilizar o `JSONPath` para localizar elementos em qualquer lugar da árvore utilizando `..` em vez de `.` (ficando em `$..legs` até o momento):

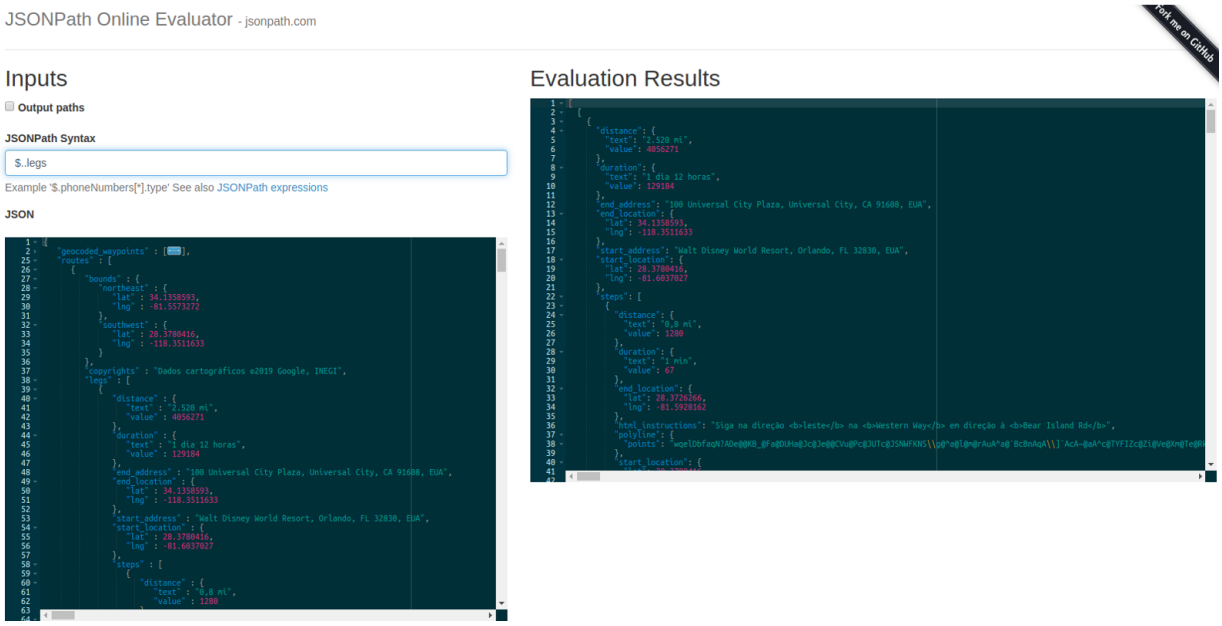


Figura 4.17: $..legs

Agora, precisamos recuperar o elemento `duration`. No entanto, se colocarmos o `JSONPath` como `$..legs.duration`, nenhum resultado é retornado. O `jsonpath.com` dá a mensagem `No match`. Por que isso acontece?

Acontece que o nosso elemento `legs` é um array. Isso significa que o site não conseguiu localizar o elemento `duration`, pois não sabia quais elementos do array buscar (ainda que haja apenas um). Para fazer com que um elemento específico seja analisado, basta colocar entre colchetes o número desse elemento (iniciando em 0). Caso queiramos que todos sejam analisados, então utilizamos um `*`. Assim, nossa *query* fica `$..legs[*].duration` até o momento.

Finalmente, precisamos do campo `value`. Basta adicionar `.value` no final da *query* e temos o resultado desejado:

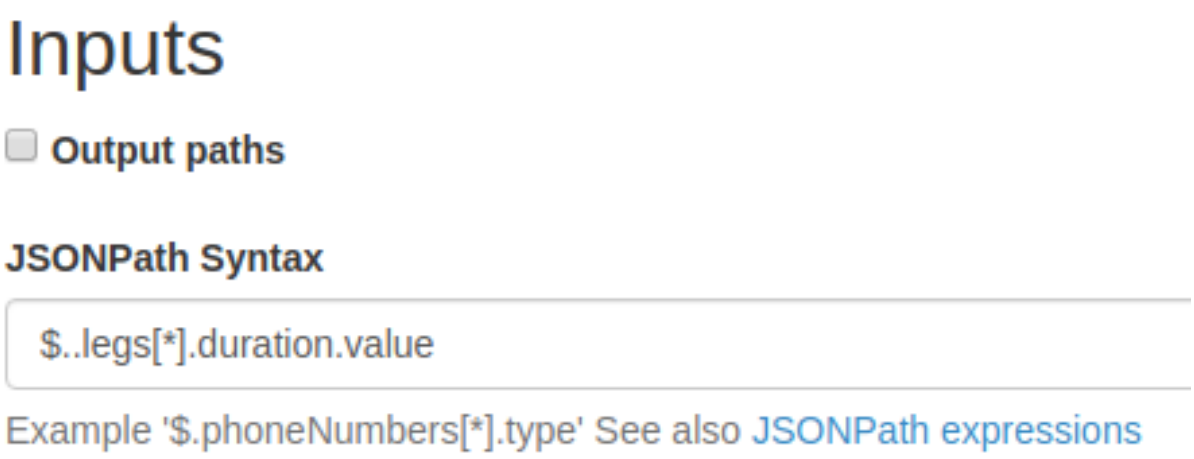


Figura 4.18: A query de antes, agora completa

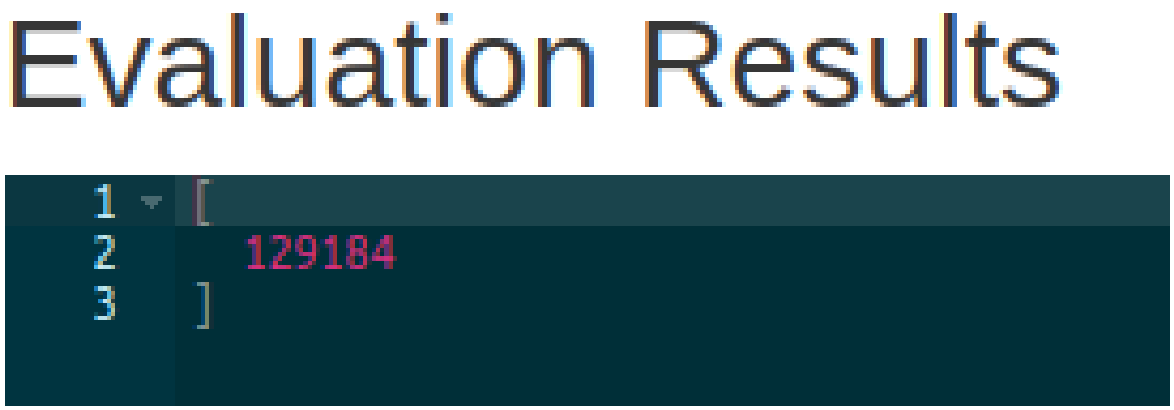


Figura 4.19: O resultado da query

Observe que o resultado que tivemos foi um array, em vez do valor direto. Por que isso acontece?

Veja que, conforme usamos `..` na nossa *query*, podemos ter um resultado de tamanho indeterminado de acordo com nosso JSON. Assim sendo, o `JSONPath` retorna esse array sempre que usamos esse elemento. Se tivéssemos usado apenas uma *query* simples, com `.`, ele traria o resultado simples.

Vamos voltar ao nosso código. Como já adicionamos a biblioteca de `JSONPath` no nosso projeto, vamos usar o método `parse` da classe `JsonPath` para analisar a `String` que já tínhamos com o JSON de que precisamos:

```
package app.car.cap04.interfaces.outcoming

import com.jayway.jsonpath.JsonPath
import org.springframework.beans.factory.annotation.Value
import org.springframework.stereotype.Service
import org.springframework.web.client.RestTemplate

@Service
class GMapsService(
    @Value("\${app.car.domain.googlemaps.apikey}")
    val appKey: String
) {
```

```
    val GMAPS_TEMPLATE: String =
"https://maps.googleapis.com/maps/api/directions/json?origin=
{origin}&destination={destination}&key={key}"

    fun getDistanceBetweenAddresses(addressOne: String, addressTwo:
String) : Int {
        val template = RestTemplate()
        val jsonResult = template.getForObject(GMAPS_TEMPLATE,
String::class.java, addressOne, addressTwo, appKey)

        JsonPath.parse(jsonResult)

        return 0
    }
}
```

Agora, podemos invocar diretamente o método `read` passando o `JSONPath` que elaboramos e atribuindo os dados a um `JSONArray`:

```
val rawResults: JSONArray =
JsonPath.parse(jsonResult).read("\$..legs[*].duration.value")
```

Vamos utilizar a API do Kotlin para converter os dados contidos no `JSONArray` em uma lista de `Ints`. Depois, vamos utilizar a API para recuperar o valor mínimo e, caso não haja valor nenhum, retornar nulo. Então, vamos utilizar o `Elvis operator` para checar se é nulo e, em caso positivo, retornar o valor máximo possível para um inteiro. Isso é feito para a possibilidade de a API ter retornado vários valores possíveis (ou seja, várias rotas), ou nenhuma rota - neste caso, vamos trazer o maior número para demonstrar a impossibilidade de se navegar de um ponto a outro:

```
val rawResults: JSONArray =
JsonPath.parse(jsonResult).read("\$..legs[*].duration.value")
return rawResults.map { it as Int }.minOrNull() ?: Int.MAX_VALUE
```

A classe, no formato final, fica assim:

```
package app.car.cap04.interfaces.outcoming
```

```
import com.jayway.jsonpath.JsonPath
import net.minidev.json.JSONArray
import org.springframework.beans.factory.annotation.Value
import org.springframework.stereotype.Service
import org.springframework.web.client.RestTemplate

@Service
class GMapsService(
    @Value("\${app.car.domain.googlemaps.apikey}")
    val appKey: String
) {

    val GMAPS_TEMPLATE: String =
        "https://maps.googleapis.com/maps/api/directions/json?origin=
{origin}&destination={destination}&key={key}"

    fun getDistanceBetweenAddresses(addressOne: String, addressTwo:
String) : Int {
        val template = RestTemplate()
        val jsonResult = template.getForObject(GMAPS_TEMPLATE,
String::class.java, addressOne, addressTwo, appKey)

        val rawResults: JSONArray =
JsonPath.parse(jsonResult).read("\$..legs[*].duration.value")
        return rawResults.map { it as Int }.minOrNull() ?: Int.MAX_VALUE
    }

}
```

4.5 Integrando a consulta ao projeto

Vamos agora criar a API de consulta às viagens que estão próximas. Para isso, vamos criar um método novo na nossa classe de API, que vai se chamar `listNearbyRequests`. Esse método vai retornar uma lista de `EntityModel<TravelRequestOutput>`, ou seja, uma lista de retornos semelhantes ao que fizemos no método `makeTravelRequest`. Esse método também vai receber como

parâmetro o endereço atual do motorista que utilizar essa API, para que possamos fazer a comparação de distância e filtrar as viagens que estão muito distantes do ponto atual.

Essa API também vai ser mapeada utilizando o método `GET` (já que estamos listando dados que vêm do servidor). Como as requisições de viagens que vamos listar por essa API encaixam-se no subconjunto das solicitações de viagens que estão próximas, vamos criar um mapeamento para que a URL fique `/travelRequests/nearby`. Além disso, também vamos incluir um `query parameter` chamado `currentAddress`, para que o usuário da API possa receber apenas as solicitações das viagens que estejam próximas desse endereço. Isso é feito através da anotação `@RequestParam`:

```
@GetMapping("/nearby")
fun listNearbyRequests(@RequestParam currentAddress: String):
    List<EntityModel<TravelRequestOutput>> {
    return emptyList()
}
```

Vamos incluir no nosso repositório um método que localize apenas `TravelRequests` que estejam no status `CREATED`. Para deixar mais genérico, um método que localize pelo `status`, ou seja, `findByStatus`:

```
interface TravelRequestRepository: JpaRepository<TravelRequest, Long> {
    fun findByStatus(status: TravelRequestStatus): List<TravelRequest>
}
```

Como o mapeamento dos métodos é feito pelo Spring Data?

Como você pôde observar, só o que fizemos para criar o mapeamento do método para o banco de dados foi chamar o método de `findByStatus`. Isso faz com que o método siga uma convenção do próprio Spring Data e, portanto, que o *framework* já faça o mapeamento automático para uma query de banco de dados. Caso não queiramos utilizar essa convenção, basta utilizar a anotação `@Query`.

Vamos concentrar a nossa lógica de mapeamento na classe `TravelService`. O nosso primeiro passo nessa classe é criar uma constante para determinar quanto tempo será o nosso corte para filtragem. Observe que o campo presente na API do Google retorna os dados em segundos; portanto, se quisermos que nosso corte seja de até dez minutos para chegada do motorista ao ponto de embarque do passageiro, vamos utilizar uma constante com o valor 600:

```
val MAX_TRAVEL_TIME: Int = 600
```

Agora, vamos injetar o serviço do Google nessa classe:

```
class TravelService(
    val travelRequestRepository: TravelRequestRepository,
    val gMapsService: GMapsService
)
```

Finalmente, vamos criar um método cujo propósito seja o de recuperar do banco de dados as solicitações de viagem cujos `status` sejam `CREATED` e, então, utilizar o serviço do Google para calcular o tempo de chegada em cada uma das solicitações, desprezando aquelas cujo tempo de chegada seja maior do que os dez minutos.

Vamos começar pela assinatura do método e recuperação dos dados do banco de dados:

```
fun listNearbyTravelRequests(currentAddress: String): List<TravelRequest>
{
    val requests =
travelRequestRepository.findByStatus(TravelRequestStatus.CREATED)
}
```

O próximo passo é realizar a filtragem de dados utilizando o `GMapsService`. Vamos repassar o parâmetro `currentAddress` para o método `getDistanceBetweenAddresses` da classe `GMapsService`, bem como o campo `origin` de cada `TravelRequest`, então separamos apenas as solicitações onde o resultado da invocação desse método é menor do que o valor presente na constante `MAX_TRAVEL_TIME`:

```
fun listNearbyTravelRequests(currentAddress: String): List<TravelRequest>
{
    val requests =
travelRequestRepository.findByStatus(TravelRequestStatus.CREATED)
    return requests.filter {
        tr -> gMapsService.getDistanceBetweenAddresses(currentAddress,
tr.origin) < MAX_TRAVEL_TIME
    }
}
```

Voltando à nossa classe `TravelRequestAPI`, vamos realizar a chamada para o método `listNearbyTravelRequests` recém-criado, para que possamos trabalhar com o resultado desse serviço:

```
@GetMapping("/nearby")
fun listNearbyRequests(@RequestParam currentAddress: String):
List<EntityModel<TravelRequestOutput>> {
    val requests = travelService.listNearbyTravelRequests(currentAddress)
    return emptyList()
}
```

Vamos relembrar que o método `buildOutputModel`, que faz a construção da classe com o `EntityModel`, recebe como parâmetros um `TravelRequest` e um `TravelRequestOutput`. Esse último é construído

a partir da chamada do método `map`, portanto precisamos chamá-lo primeiro e, na sequência, chamar o método `buildOutputModel` com ambos os parâmetros. Para simplificar essas chamadas, vamos criar um novo método na classe `TravelRequestMapper` que vai receber a lista de `TravelRequest` e vai chamar, para cada uma, o método `map` e depois o `buildOutputModel` - dessa forma, retornando uma lista de `EntityModel<TravelRequestOutput>`. Vamos chamar esse método de `buildOutputModel` (ou seja, estamos sobrecarregando o método que já existia):

```
fun buildOutputModel(requests: List<TravelRequest>) =
    requests.map { buildOutputModel(it, map(it)) }
```

> **Por que o método novo referenciou uma variável chamada it?**
>
> Em Kotlin, várias lambdas adotam por convenção o seguinte cenário: caso a variável sobre a qual a função está operando não receba um nome específico, ela é referenciada através do nome `it`. O método `filter`, por exemplo, adota essa convenção, além de métodos, como `let`, `also` etc.

Voltando à nossa classe `TravelRequestAPI`, vamos fazer com que ela retorne o resultado da chamada ao método recém-criado:

```
@GetMapping("/nearby")
fun listNearbyRequests(@RequestParam currentAddress: String):
List<EntityModel<TravelRequestOutput>> {
    val requests = travelService.listNearbyTravelRequests(currentAddress)
    return mapper.buildOutputModel(requests)
}
```

4.6 Testando a nova API

O último passo é fazer o teste da nova API. Para isso, vamos recriar o passageiro:

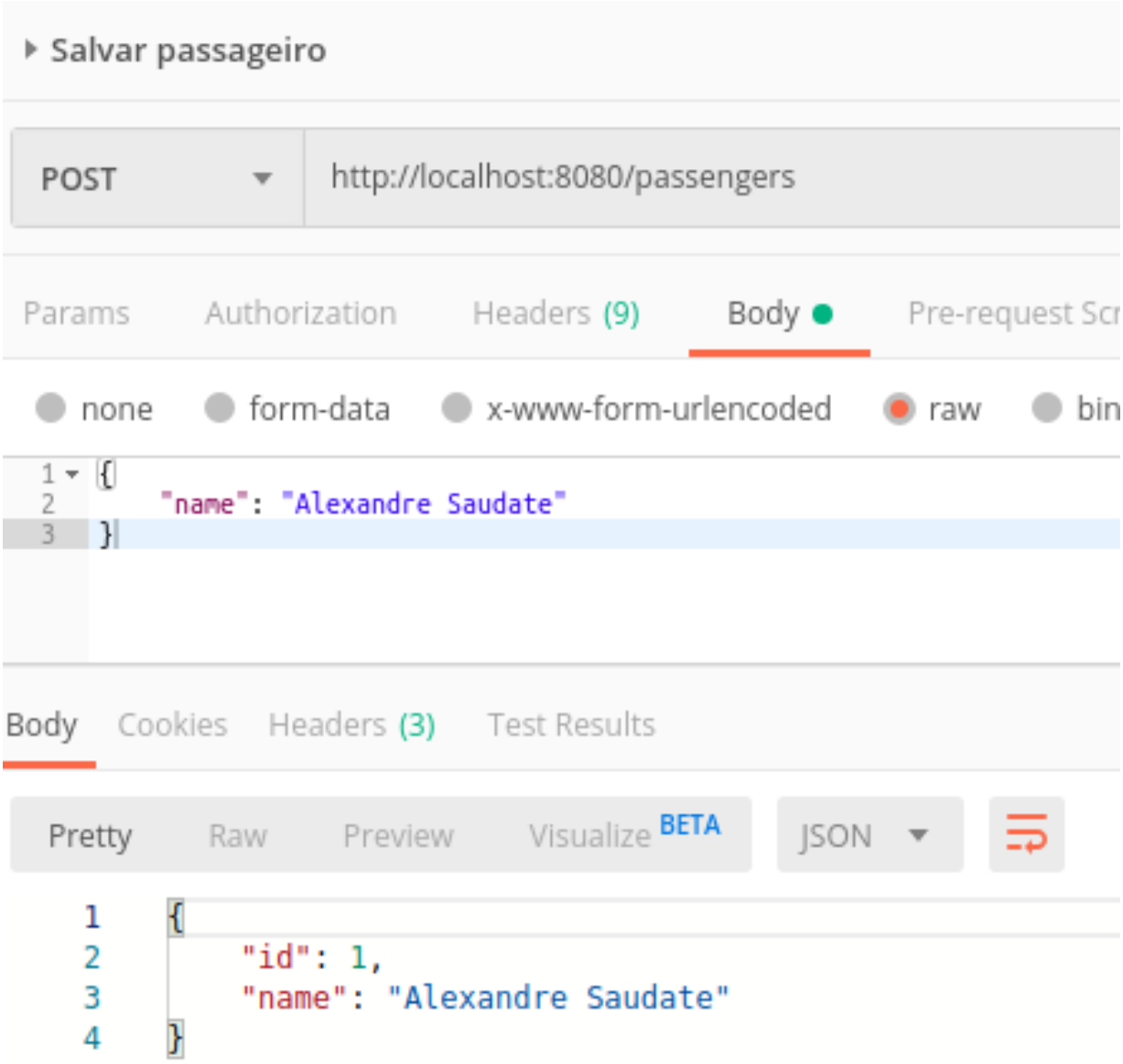


Figura 4.20: Passageiro recriado

Depois, vamos criar a solicitação de viagem:

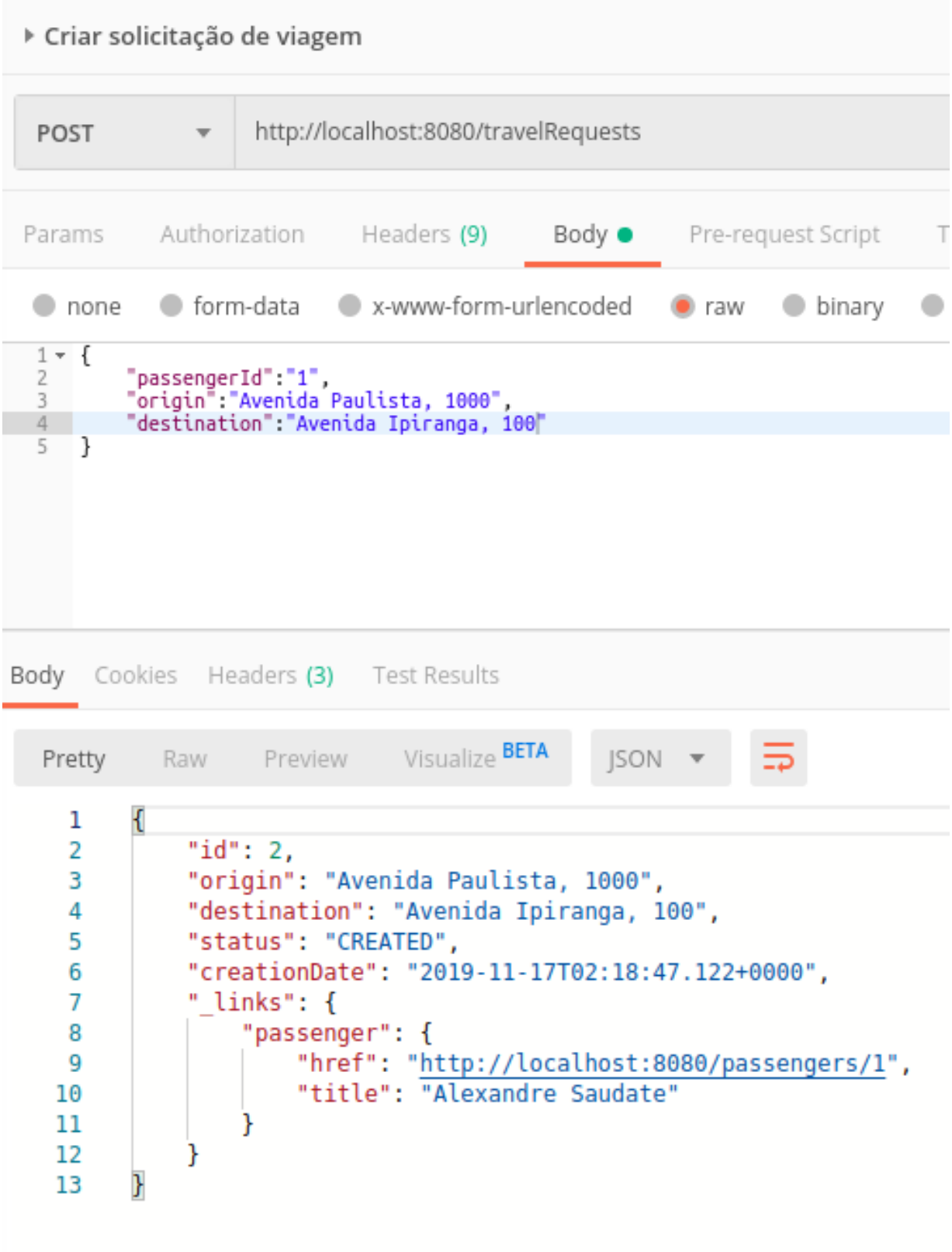


Figura 4.21: Solicitação de viagem criada

Finalmente, vamos realizar a listagem de corridas próximas:

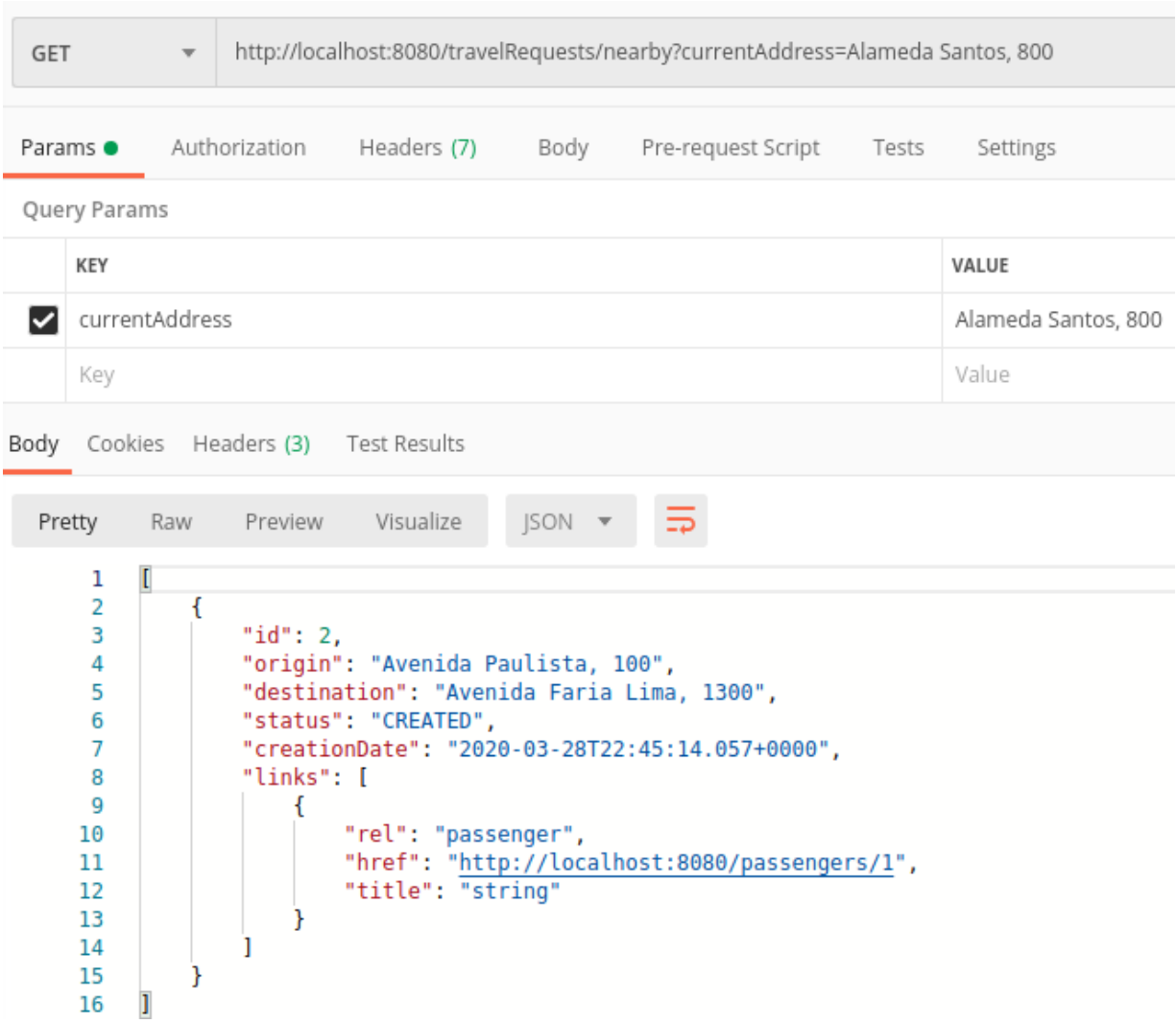


Figura 4.22: Retorno da viagem que está próxima

Observe que, caso um endereço mais distante seja passado como parâmetro, a corrida não é retornada:

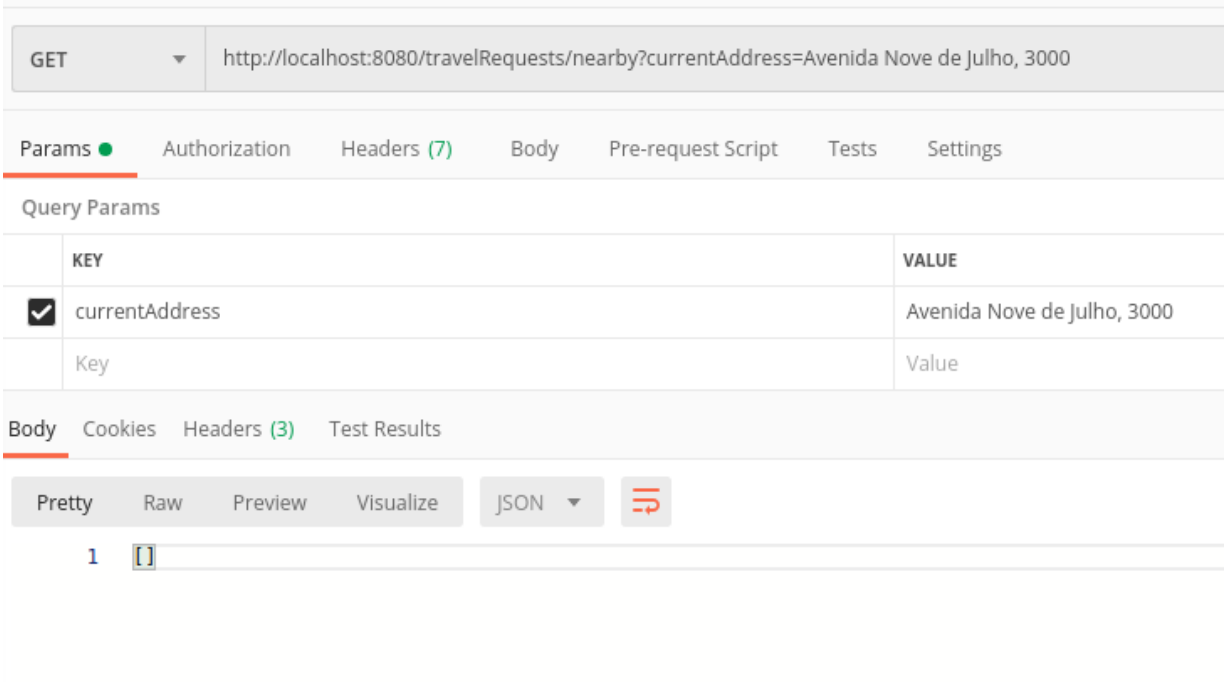


Figura 4.23: Ausência de resultados com endereços distantes

Conclusão

Neste capítulo, conhecemos um pouco sobre como funciona a parte de criação de clientes REST utilizando o RestTemplate do Spring Boot. Obviamente, criamos um tipo de cliente superficial que consome um serviço externo, mas que nos dá insumos para conhecermos melhor um aspecto muito importante da criação de serviços REST, a criação de testes automatizados para os nossos serviços.

CAPÍTULO 5
Criando os testes automatizados

"No meio do caos há sempre uma oportunidade." - Sun Tzu.

Nossa API já oferece, efetivamente, uma funcionalidade: é possível que um motorista liste quais solicitações de viagens foram feitas próximas a ele. Ainda não criamos a API que associa a este motorista uma determinada viagem, mas, com uma funcionalidade já criada, é muito importante que comecemos a criar **testes automatizados** para ela, pois são eles que vão garantir a estabilidade quando fizermos quaisquer mudanças.

5.1 Conhecendo as estratégias de teste

Como desenvolvedor, talvez você já esteja familiarizado com algumas técnicas de testes dentro do próprio ambiente, como testes unitários e de integração, por exemplo. Os testes unitários restringem-se a um único componente e suas dependências são fornecidas como *mocks*, ou seja, retornam respostas pré-programadas. Já os testes de integração abrangem escopos mais amplos, atingindo vários componentes, algumas vezes também requerendo *mocks*, mas procurando depender o mínimo possível deles.

Quando partimos para testes de APIs, elevamos esses testes a um novo patamar com os **testes de contrato**. Os testes de contrato são como os testes de integração, mas são feitos testando inclusive a interface da API. Isso quer dizer que, se um teste de integração pode ser feito instanciando a classe Kotlin que fornece a API e realizando uma chamada para o método, um teste de contrato vai requerer *start* no servidor e realizar uma chamada HTTP a ele. Isso

garante que não apenas o funcionamento do sistema esteja correto, mas também que o sistema esteja realizando o mapeamento correto dos formatos de entrada e saída - daí o nome **contrato**.

O Spring Boot, por si só, não realiza esse tipo de teste. Portanto, podemos contar com um novo framework, o REST Assured (disponível em http://rest-assured.io/). Ele atua em conjunto com o Spring Boot para fornecer testes de APIs muito poderosos. Vamos criar os testes da API de passageiros primeiro para verificar como isso funciona.

5.2 Criando os testes da API de passageiros com REST Assured

Em primeiro lugar, precisamos adicionar a dependência do REST Assured no nosso `build.gradle.kts`. Vamos utilizar também um framework chamado JUnit (que é o padrão *de fato* para testes em Java e Kotlin), que já foi incluído no projeto pelo Spring Initializr no ato da criação do projeto.

A versão mais atual do REST Assured (no momento da escrita deste livro) é a 4.3.2. Vamos incluir então o trecho correspondente no `build.gradle.kts`, com o escopo de testes:

```
testImplementation("io.rest-assured:spring-mock-mvc:4.3.2")
```

Agora, vamos criar a classe de testes no diretório padrão de testes, que é `src/test/kotlin`. Nele, vamos criar o pacote `app.car.cap05.interfaces.incoming`, que é o mesmo pacote da classe `PassengerAPI`, que fornece a nossa API de passageiros. Nossa classe se chamará `PassengerAPITestIT`, ou seja, o nome da classe a ser testada acrescido do sufixo `TestIT`, que é uma forma de declarar esse teste como um teste de integração. Seu código começa assim:

```
package app.car.cap05.interfaces.incoming

class PassengerAPITestIT {
}
```

Quando os testes forem executados por essa classe, a tarefa dela é inicializar todo o contexto do Spring, assim como o *web server* associado. Isso pode ser feito através de uma única anotação, a `@SpringBootTest`:

```
import org.springframework.boot.test.context.SpringBootTest

@SpringBootTest
class PassengerAPITestIT {
}
```

Tendo isso em mente, vamos considerar que os testes são executados em um ambiente CI/CD (*Continuous Integration*/*Continuous Delivery*). Isso significa que pode haver vários testes desse sistema sendo executados em paralelo. Sendo assim, como proceder para que uns não bloqueiem os outros por ocuparem a mesma porta de rede quando o servidor for inicializado? Lembre-se de que, por padrão, o sistema sempre é executado sob a porta 8080.

A resposta para esse problema é atribuir uma porta **aleatória** toda vez que o sistema for inicializado. Isso quer dizer que o Spring Boot é capaz de procurar por uma porta vazia no ambiente e associar o *web server* a ela. Isso pode ser feito utilizando o parâmetro `webEnvironment` da anotação `@SpringBootTest`, passando como parâmetro a constante `RANDOM_PORT`, da forma como segue:

```
import org.springframework.boot.test.context.SpringBootTest

@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class PassengerAPITestIT {
}
```

Para que o REST Assured saiba qual porta utilizar para realizar as requisições, utilizamos um artifício do Spring Boot, que é a anotação `@LocalServerPort`. Essa anotação realiza a injeção da porta em um campo da classe de testes, assim:

```
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.web.server.LocalServerPort

@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class PassengerAPITestIT {

    @LocalServerPort
    private var port: Int = 0

}
```

Para configurar o REST Assured com essa porta, simplesmente realizamos a atribuição desse valor injetado no campo `port` da classe `RestAssured` de forma estática. Vamos criar um método no qual incluiremos essa informação, e anotá-lo com a anotação `@BeforeEach` do JUnit, que fará com que esse método sempre seja invocado antes da execução de cada teste:

```
import io.restassured.RestAssured
import org.junit.jupiter.api.BeforeEach
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.web.server.LocalServerPort

@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class PassengerAPITestIT {

    @LocalServerPort
    private var port: Int = 0

    @BeforeEach
    fun setup() {
        RestAssured.port = port
```

```
    }
}
```

Vamos começar a criação do nosso teste. Para isso, vamos criar um método com um nome apropriado para dizer o que, exatamente, estamos testando. Depois, vamos anotá-lo com a anotação `@Test` do JUnit:

```
package app.car.cap05.interfaces.incoming

import io.restassured.RestAssured
import org.junit.jupiter.api.BeforeEach
import org.junit.jupiter.api.Test
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.web.server.LocalServerPort

@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class PassengerAPITestIT {

    @LocalServerPort
    private var port: Int = 0

    @BeforeEach
    fun setup() {
        RestAssured.port = port
    }

    @Test
    fun testCreatePassenger() {

    }
}
```

Vamos agora criar uma String contendo o JSON que usamos para criação de um passageiro. Podemos utilizar três aspas duplas (`"`) seguidas para criar uma string multilinhas em Kotlin:

```
@Test
fun testCreatePassenger() {
    val createPassengerJSON = """
```

```
        {"name":"Alexandre Saudate"}
    """.trimIndent()
}
```

Passemos à criação do teste da API. Primeiro vamos reunir as informações acerca do básico de uma API:

- **Tipo de conteúdo**: JSON
- **URL**: `/passengers`
- **Método HTTP**: `POST`
- **Código de retorno esperado**: `200`

Com essas informações, podemos criar o teste utilizando a API do REST Assured. É importante observar que ela segue o conceito de uma API Fluente (como apresentado por FOWLER, 2005), e é interessante, nesse caso, realizar importação estática do primeiro método, `given` . Esse método vai inicializar a declaração de uma API com o REST Assured:

```
package app.car.cap05.interfaces.incoming;

import io.restassured.RestAssured.given

// restante do código omitido

@Test
fun testCreatePassenger() {
    val createPassengerJSON = """
        {"name":"Alexandre Saudate"}
    """.trimIndent()

    given()
}
```

Agora, vamos inserir a declaração do tipo de conteúdo, utilizando na sequência o método `contentType` e passando como parâmetro o valor `JSON` , presente na enumeração `ContentType` :

```
given()
    .contentType(io.restassured.http.ContentType.JSON)
```

Depois, vamos realizar a atribuição do corpo da requisição utilizando o método `body`:

```
val createPassengerJSON = """
        {"name":"Alexandre Saudate"}
    """.trimIndent()

given()
    .contentType(io.restassured.http.ContentType.JSON)
    .body(createPassengerJSON)
```

Na sequência, especificamos de uma só vez o método HTTP e a URL. Como o método é `POST`, utilizamos o método `post`:

```
given()
   .contentType(io.restassured.http.ContentType.JSON)
   .body(createPassengerJSON)
   .post("/passengers")
```

Até aqui, fornecemos ao REST Assured todo o necessário para a realização da requisição. Agora vamos começar a seção de asserções sobre o que foi executado, e para realizar a separação entre o trecho da requisição e validação da resposta utilizamos o método `then`:

```
given()
   .contentType(io.restassured.http.ContentType.JSON)
   .body(createPassengerJSON)
   .post("/passengers")
   .then()
```

Para verificarmos o código de retorno, utilizamos o método `statusCode`:

```
given()
   .contentType(io.restassured.http.ContentType.JSON)
   .body(createPassengerJSON)
   .post("/passengers")
   .then()
   .statusCode(200)
```

Finalmente, vamos nos certificar de que o JSON de resposta contém os campos de que precisamos, ou seja, `id` e `name`. Essa verificação é feita pelo método `body`, que recebe como parâmetros uma String `GPath` (muito semelhante ao `JSONPath`) e um *matcher* de verificação de um framework de apoio, o Hamcrest. Esse *matcher* vai ser responsável por verificar que o dado resultante da recuperação feita pelo `GPath` é igual ao que necessitamos. Nesse caso, vamos usar os *matchers* `equalTo` e o `notNullValue`. Realizaremos a importação estática dos *matchers* e, então, executaremos o método:

```
import org.hamcrest.Matchers.equalTo;
import org.hamcrest.Matchers.notNullValue;

// Restante do código omitido...

given()
   .contentType(io.restassured.http.ContentType.JSON)
   .body(createPassengerJSON)
   .post("/passengers")
   .then()
   .statusCode(200)
   .body("id", notNullValue())
   .body("name", equalTo("Alexandre Saudate"))
```

POR QUE REALIZAMOS A VERIFICAÇÃO DO CAMPO ID COM `NOTNULLVALUE` EM VEZ DE `EQUALTO`?

Com a execução de vários testes no projeto, pode ser que nem sempre o valor de um ID seja igual ao que codificamos no teste. Assim sendo, fazemos a verificação apenas de que o campo está presente, em vez de compará-lo com um valor específico.

5.3 Executando o teste

Quando criamos o teste com o sufixo `IT`, estamos adotando um padrão de nomenclatura que é utilizado para testes de integração. Ainda que estejamos usando um teste de contrato, há uma coisa em comum entre eles: o tempo de execução. Quando executo o teste na minha IDE, recebo como resposta algo semelhante ao seguinte:

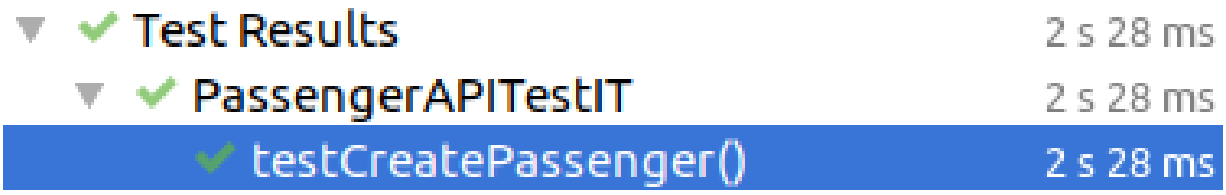


Figura 5.1: Resultado da execução do teste na IDE

Em minha máquina, o tempo de execução foi de 2 segundos e 28 milissegundos. Apesar de ser um tempo relativamente rápido, se tivermos dez testes para o nosso sistema que levem o mesmo tempo, esse tempo de execução vai subir para um pouco mais de 20 segundos - o que já começa a ser impraticável. Mike Wacker (WACKER, 2015), na época da escrita do artigo *Just Say No To More End-To-End Tests*, sinalizou a sugestão do Google para testes: 70% de testes unitários, 20% de testes de integração e 10% de testes *end-to-end*. Testes de contrato não são mencionados no artigo, mas esses são um tipo intermediário entre os testes de integração e os testes *end-to_end. A diferença é que os testes _end-to-end* não usam *mocks* em ponto algum e os de contrato, sim.

Para rodar os testes, basta executar na raiz do projeto o comando `./gradlew test`. Isso vai fazer o Gradle compilar todo o projeto e então executar os testes (tanto unitários quanto integrados), e algo semelhante ao seguinte deve aparecer entre os resultados da execução:

```
BUILD SUCCESSFUL in 19s
5 actionable tasks: 5 executed
```

O Gradle não deixa muito claro os resultados dos testes no console. Mas basta abrir o arquivo `build/reports/tests/test/index.html` no

browser que podemos verificar o seguinte:

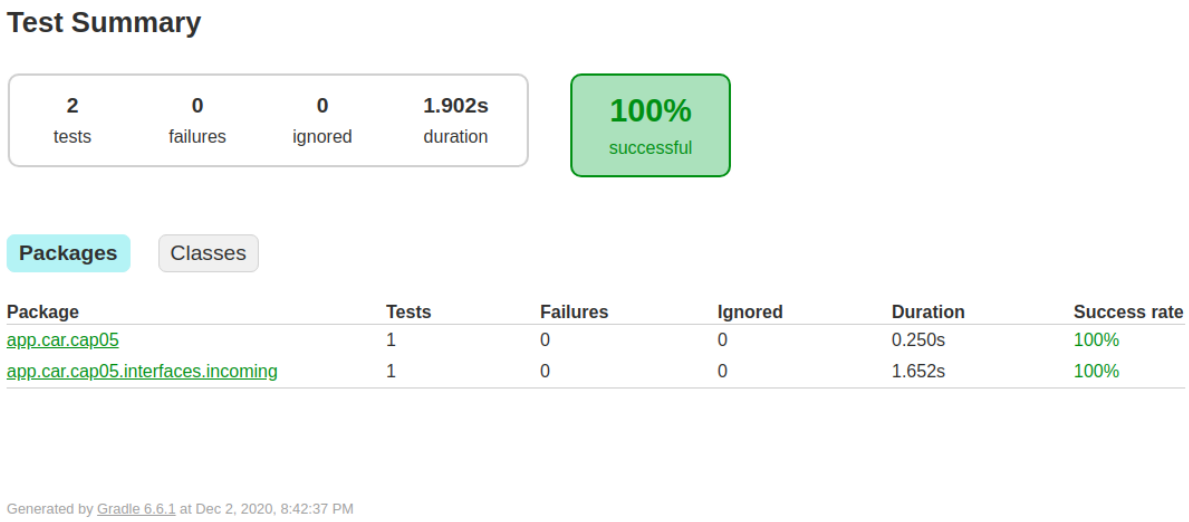


Figura 5.2: Resultado da execução dos testes pelo Gradle

5.4 Testes mais completos com WireMock

Nossa API de passageiros foi mais simples de testar, porque não envolvia serviços externos. Já o serviço de consulta de solicitações de viagens é um pouco mais complexo, pois ele faz a consulta ao serviço do Google para checar o tempo de viagem. Mas não podemos invocar o serviço a cada execução do teste, pois isso eventualmente pode ser cobrado. Há também casos em que não é possível invocar um serviço externo por haver efeitos colaterais, como serviços de cobrança ou serviços mais complexos como ERPs. Assim sendo, a melhor estratégia seria criar um *mock* para esses serviços externos - mas que fosse algo tão próximo do real quanto possível. Em outras palavras, o ideal seria criar um serviço que fornecesse uma API muito próxima do serviço real, mas que fosse controlada por nós.

Para atender a esse propósito, existe um framework chamado WireMock (http://wiremock.org/). Ele utiliza uma linguagem fluente

para que possamos criar um *mock server* capaz de atuar como um dublê do serviço que vai ser consultado na outra ponta - no caso, o do Google Maps.

O Spring Boot oferece uma integração nativa com o WireMock, que realiza o gerenciamento automático do ciclo de vida do *mock server*. Para incluí-lo no nosso projeto, temos que incluir a seguinte entrada no `build.gradle.kts`:

```
testImplementation("org.springframework.cloud:spring-cloud-contract-wiremock:2.2.5.RELEASE")
```

Como nós vamos criar esse dublê, o endereço dele será diferente do original do Google Maps. O Spring Boot oferece uma estratégia para lidar com esse tipo de cenário a partir do uso de *profiles*. Criar um *profile* no Spring Boot significa criar um novo arquivo no padrão `application-(nome do profile)` e, então, populá-lo com os dados que a aplicação vai utilizar no cenário específico.

No nosso caso, vamos modificar o serviço `GMapsService` para que seja possível alterar o *host* da API. Assim sendo, vamos criar um atributo nessa classe que identifique esse *host* e já seja populado por padrão com o endereço do Google, ao mesmo tempo em que também pode ser sobrescrito com o valor definido por algum *profile*. Isso pode ser feito utilizando a anotação `@Value`, declarando a chave e o valor padrão, separados por `:` e delimitados por `{` e `}`. Vamos dizer ao Spring Boot que ele deve preencher o atributo com o valor da chave `interfaces.outcoming.gmaps.host` e, caso não encontre, popular com o valor padrão `https://maps.googleapis.com`. Através da técnica de interpolação de strings, vamos inserir o valor da variável na constante `GMAPS_TEMPLATE`. O trecho de código fica assim:

```
@Service
class GMapsService(
    @Value("\${app.car.domain.googlemaps.apikey}")
    val appKey: String,
```

```
@Value("\${interfaces.outcoming.gmaps.host:https://maps.googleapis.com}")
    val gMapsHost: String
) {

    val GMAPS_TEMPLATE: String = "$gMapsHost/maps/api/directions/json?
origin={origin}&destination={destination}&key={key}"

    fun getDistanceBetweenAddresses(addressOne: String, addressTwo:
String) : Int {
        val template = RestTemplate()
        val jsonResult = template.getForObject(GMAPS_TEMPLATE,
String::class.java, addressOne, addressTwo, appKey)
        // código restante omitido
    }
}
```

Agora, vamos realizar a criação da classe de testes propriamente dita. A estrutura geral dessa classe será semelhante à `PassengerAPITestIT`, inclusive com a criação de uma porta aleatória para os testes:

```
@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
class TravelRequestAPITestIT {

    @LocalServerPort
    private var port:Int = 0

    @BeforeEach
    fun setup() {
        RestAssured.port = port
    }
}
```

Vamos incluir a anotação `@AutoConfigureWireMock` nesta classe, que vai fazer com que o WireMock seja criado automaticamente:

```
// Restante dos imports omitidos
import org.springframework.cloud.contract.wiremock.AutoConfigureWireMock

@SpringBootTest(webEnvironment =
```

```
SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock
class TravelRequestAPITestIT {
```

Da forma como está, o `WireMock` será inicializado tomando como base a porta 8080 (o que pode gerar um conflito com a nossa própria aplicação ou com execuções paralelas do teste). Podemos fazer com que ele seja inicializado em uma porta aleatória utilizando a propriedade `port` da anotação `@AutoConfigureWireMock`, passando como parâmetro a constante `DYNAMIC_PORT`:

```
import com.github.tomakehurst.wiremock.core.WireMockConfiguration;

// Restante dos imports omitidos

@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = WireMockConfiguration.DYNAMIC_PORT)
class TravelRequestAPITestIT {
```

Vamos inserir um atributo nessa classe que vai corresponder ao servidor do WireMock, do tipo `WireMockServer`. Dessa vez, não poderemos declarar a dependência no construtor da classe devido a uma incompatibilidade com o JUnit. Dessa forma, precisamos informar ao Kotlin que essa variável será inicializada após a instanciação da classe, usando o modificador `lateinit`:

```
import org.springframework.beans.factory.annotation.Autowired
import com.github.tomakehurst.wiremock.WireMockServer

class TravelRequestAPITestIT{

    @Autowired
    lateinit var server: WireMockServer

    // Restante do código omitido
```

Agora, precisamos configurar um *profile* especial de testes, para que, ao ser executado, o teste use o endereço do WireMock, em vez do endereço real. Dentro do diretório `src/test/resources`, crie um

arquivo chamado `application-test.properties` (dessa forma, o nome do *profile* será `test`). Ele deverá ter o seguinte conteúdo:

```
interfaces.outcoming.gmaps.host=http://localhost:${wiremock.server.port}
```

> **O QUE ACONTECE COM A CHAVE DO GOOGLE NO PROFILE DE TESTES?**
>
> O Spring conta com uma estrutura de encadeamento de valores em seus *profiles*. Como o arquivo que já tínhamos não tem um nome de *profile* especificado, ele assume que todos os valores declarados naquele arquivo são os padrões. Portanto, não precisamos redeclarar os atributos pois o Spring entende que aqueles valores apenas não foram sobrescritos.

Finalmente, precisamos modificar o teste para que ele use o novo *profile*. Isso é feito através da anotação `@ActiveProfiles`, que recebe como parâmetro o nome do *profile* que estará ativo (no caso, `test`):

```
//Restante dos imports omitidos
import org.springframework.test.context.ActiveProfiles

@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = WireMockConfiguration.DYNAMIC_PORT)
@ActiveProfiles("test")
class TravelRequestAPITestIT{

    // Restante do código omitido
```

5.5 Configuração do mock do Google Maps

O próximo passo é criar o dublê do Google Maps. Para isso, vamos capturar mais uma vez a resposta real do serviço utilizando nosso *browser* para recuperar a distância da Avenida Paulista, 900 para a

Avenida Paulista, 1000. O endereço será, portanto, https://maps.googleapis.com/maps/api/directions/json?origin=Avenida%20Paulista,%20900&destination=Avenida%20Paulista,%201000&key=APIKEY. Observe que a resposta é demasiado grande para colocar em uma String, então vamos colocar em um arquivo e ler a partir dele. Na pasta de fontes de recursos de testes (`src/test/resources`), crie a pasta `responses/gmaps` e, dentro dela, o arquivo `sample_response.json` . Copie para esse arquivo a resposta do serviço do Google Maps.

Vamos criar agora um arquivo chamado `FileUtils.kt` , que vai fornecer métodos utilitários para leitura de arquivos. Dentro desse arquivo, vamos criar uma função chamada `loadFileContents` , que vai receber como parâmetro o nome do arquivo a ser carregado. Quando carregamos arquivos do projeto, em geral temos que lidar com complicações em relação à mudança de posição do arquivo dependendo do formato do projeto (se já está compactado em um JAR ou não, se a posição do arquivo é alterada após a compilação etc.). Para não termos que lidar com essas complicações, utilizamos a classe `ClassPathResource` do Spring, onde passamos uma String com o nome do arquivo no construtor. Depois de instanciado, basta invocar `.inputStream` para recuperar uma `InputStream` com os dados do arquivo:

```
package app.car.cap05.infrastructure

import org.springframework.core.io.ClassPathResource

fun loadFileContents(fileName: String) =
ClassPathResource(fileName).inputStream
```

O último ponto da criação desse método é a leitura da *stream*. Para fazer isso, podemos invocar o método `readAllBytes` da `InputStream` e então, converter o conteúdo em uma String usando o método `toString` , passando o *charset* como parâmetro. Vamos utilizar o *charset* `UTF-8` para oferecer o máximo de compatibilidade com o ambiente. A função fica assim, portanto:

```
fun loadFileContents(fileName: String) =

ClassPathResource(fileName).inputStream.readAllBytes().toString(Charsets.U
TF_8)
```

Para criar nosso dublê, precisamos separar toda a informação necessária. O serviço do Google Maps original vai até uma URL chamada `/maps/api/directions/json` para obter os dados, e recebe os parâmetros através das *query strings* `origin`, `destination` e `key`. Ao receber uma solicitação nessa URL, e com os parâmetros de cada *query string* com o valor correto, o servidor do WireMock deve retornar uma resposta de sucesso com o JSON contido no arquivo `sample_response.json`, que já populamos com a resposta real do serviço.

Para realizar toda a configuração desse servidor, vamos realizar a importação do conteúdo da classe `com.github.tomakehurst.wiremock.client.WireMock` e criar um método separado chamado `setupServer`. Feito isso, vamos chamar o método `stubFor` do servidor do WireMock, iniciando o processo de criação do servidor. Para associar a URL ao método `GET`, vamos invocar o método `get` passando como parâmetro o resultado do método `urlPathEqualTo`. Esse método, por sua vez, vai receber a URL `/maps/api/directions/json` como parâmetro:

```
import com.github.tomakehurst.wiremock.client.WireMock.*

fun setupServer() {
    server.stubFor(get(urlPathEqualTo("/maps/api/directions/json")))
}
```

Ainda não terminamos a configuração. Logo após o método `get` retornar, vamos usar o método `withQueryParam` para declarar quais *query params* desejamos receber e quais valores. Esse método recebe dois parâmetros: o primeiro é o nome do *query param* em si e o segundo, um *matcher* semelhante ao do Hamcrest para comparar a *query string*. No caso, vamos utilizar o *matcher*

chamado `equalTo` para declarar que esperamos receber valores exatamente iguais aos declarados:

```
server.stubFor(get(urlPathEqualTo("/maps/api/directions/json"))
    .withQueryParam("origin", equalTo("Avenida Paulista, 900"))
    .withQueryParam("destination", equalTo("Avenida Paulista, 1000"))
    .withQueryParam("key", equalTo("APIKEY"))
)
```

Observe que o valor da chave de API do Google está como `APIKEY`. Para que o código de teste utilize essa chave, basta editar o valor no arquivo de propriedades de teste, ou seja, no `application-test.properties`. Insira o seguinte conteúdo no arquivo:

```
interfaces.outcoming.gmaps.host=http://localhost:${wiremock.server.port}
app.car.domain.googlemaps.apikey=APIKEY
```

O último passo é realizar a declaração do retorno. O WireMock oferece um atalho para respostas de sucesso com JSON através do método `okJson`. Esse método recebe como parâmetro uma String, que vamos carregar a partir da função `loadFileContents` com o conteúdo do arquivo disponível em `/responses/gmaps/sample_response.json`. O resultado será fornecido para o método `willReturn`, que é a declaração final do conteúdo que será retornado por este *mock*. O conjunto fica assim:

```
import app.car.cap05.infrastructure.loadFileContents

server.stubFor(get(urlPathEqualTo("/maps/api/directions/json"))
    .withQueryParam("origin", equalTo("Avenida Paulista, 900"))
    .withQueryParam("destination", equalTo("Avenida Paulista, 1000"))
    .withQueryParam("key", equalTo("APIKEY"))

.willReturn(okJson(loadFileContents("/responses/gmaps/sample_response.json
")))
)
```

Finalmente, vamos realizar a criação dos testes necessários para verificar que tudo está funcionando. Faremos isso em um novo método chamado `testFindNearbyTravelRequests`. Dentro desse método,

criaremos três novas invocações com o REST Assured: uma para a criação de um novo passageiro, outra para criar uma nova solicitação de viagem e a última para recuperar as viagens próximas. Uma única modificação que faremos nos testes em relação aos que foram feitos na classe `PassengerAPITestIT` são os *matchers* do Hamcrest para verificação de igualdade. Observe que utilizamos um método chamado `equalTo` do WireMock para criação do servidor e, portanto, não podemos utilizar o mesmo *matcher* no Hamcrest com a importação simples. Para contornar essa questão, vamos utilizar um artifício do Kotlin para renomear o import. Vamos renomear o método `equalTo` do Hamcrest para `equalToHamcrest`:

```
import org.hamcrest.Matchers.equalTo as equalToHamcrest
```

Também vamos aproveitar o método `loadFileContents` para criar alguns arquivos com o conteúdo das requisições. Crie um diretório `requests/passengers_api` e, dentro dele, um arquivo chamado `create_new_passenger.json` com o seguinte conteúdo:

```
{
  "name": "Alexandre Saudate"
}
```

Agora, dentro do diretório `requests` crie um novo diretório chamado `travel_requests_api` e, dentro dele, um arquivo chamado `create_new_request.json`. Esse arquivo deverá ter o seguinte conteúdo:

```
{
  "passengerId":"1",
  "origin":"Avenida Paulista, 1000",
  "destination":"Avenida Ipiranga, 100"
}
```

Vamos à criação dos testes. Dentro do método `testFindNearbyTravelRequests` vamos incluir uma chamada ao método `setupServer` e depois, colocar uma cópia bastante semelhante ao método de teste da classe `PassengerAPITestIT`. Observe que a carga

do conteúdo da chamada será feita a partir do método `loadFileContents`:

```
@Test
fun testFindNearbyTravelRequests() {
    setupServer()
    given()
        .contentType(ContentType.JSON)

.body(loadFileContents("/requests/passengers_api/create_new_passenger.json
"))
        .post("/passengers")
        .then()
        .statusCode(200)
        .body("id", notNullValue())
        .body("name", equalToHamcrest("Alexandre Saudate"))
}
```

Faremos uma chamada semelhante ao método de criação de solicitações de viagens. A API está disponível na URL `/travelRequests`, recebe um JSON a partir do método `POST` e retorna o código de status 200. Também podemos realizar a validação de dados de forma bastante semelhante ao teste anterior:

```
given()
    .contentType(ContentType.JSON)

.body(loadFileContents("/requests/travel_requests_api/create_new_request.j
son"))
    .post("/travelRequests")
    .then()
    .statusCode(200)
    .body("id", notNullValue())
    .body("origin", equalToHamcrest("Avenida Paulista, 1000"))
    .body("destination", equalToHamcrest("Avenida Ipiranga, 100"))
    .body("status", equalToHamcrest("CREATED"))
    .body("_links.passenger.title", equalToHamcrest("Alexandre Saudate"))
```

Agora vamos criar o teste da solicitação de viagens em si. É uma API disponível na URL `/travelRequests/nearby`, que recebe um *query*

param chamado `currentAddress` através do método `GET`. Essa API retorna o código de status 200 com um array. Por conta desse fator, vamos adicionar no começo de cada string o valor `[0]`, que será equivalente ao primeiro elemento do array de resposta. Exceto por esse detalhe, o restante das validações será essencialmente o mesmo:

```
given()
    .get("/travelRequests/nearby?currentAddress=Avenida Paulista, 900")
    .then()
    .statusCode(200)
    .body("[0].id", notNullValue())
    .body("[0].origin", equalToHamcrest("Avenida Paulista, 1000"))
    .body("[0].destination", equalToHamcrest("Avenida Ipiranga, 100"))
    .body("[0].status", equalToHamcrest("CREATED"))
```

Por agora, os testes já estarão funcionando. Podemos verificar isso como fizemos antes, executando a task `test` do Gradle e depois abrindo o relatório no browser:

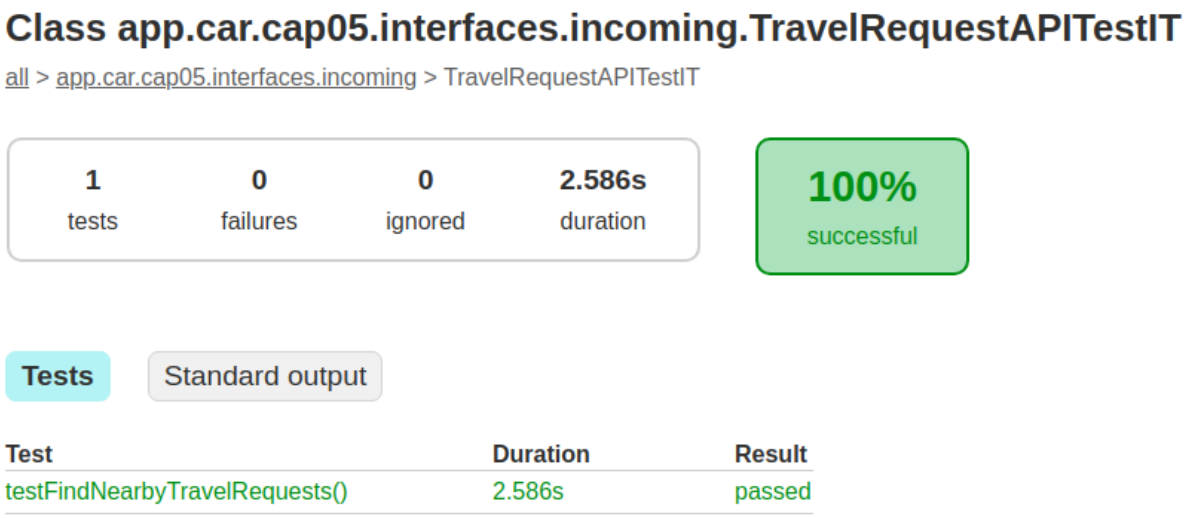


Figura 5.3: Resultado da execução dos testes pelo Gradle

Para deixar nossos testes mais confiáveis, vamos fazer com que os testes da API de solicitação de viagens utilizem os dados que retornam da criação do passageiro, em vez de utilizar o ID

hardcoded. Para isso, vamos utilizar uma técnica conhecida como *placeholders* - ou seja, declarar trechos dentro do arquivo em que são feitos com o propósito de serem modificados. Para criar um *placeholder*, vamos abrir novamente o arquivo `create_new_request.json` e alterá-lo para que o ID do passageiro seja a String `{{passengerId}}` :

```
{
  "passengerId":"{{passengerId}}",
  "origin":"Avenida Paulista, 1000",
  "destination":"Avenida Ipiranga, 100"
}
```

Em seguida, vamos alterar o método `loadFileContents` para receber um mapa como parâmetro (com o valor padrão sendo um mapa vazio). Esse mapa conterá os valores que devem ser utilizados no lugar dos *placeholders*. Dessa forma, nosso método deverá ser alterado de forma que deverá declarar o retorno como uma String, para que possamos invocar a função `forEach` do mapa. Essa função vai modificando o valor da String que contém o conteúdo do arquivo a cada interação. Além disso, também vamos usar interpolação de strings para que não seja necessário declarar as chaves de abertura e fechamento dos *placeholders* nos valores do mapa. Esse método vai chamar o método `loadFileContents` original e depois executar um laço onde, para cada chave do mapa encontrada no arquivo, será feita uma reposição com seu respectivo valor. Dessa forma, o método refatorado vai ficar assim:

```
fun loadFileContents(fileName: String, variables: Map<String,String> =
emptyMap()): String {
    var fileContents = ClassPathResource(fileName).inputStream
        .readAllBytes().toString(Charsets.UTF_8)

    variables.forEach {
        key, value -> fileContents = fileContents.replace("{{$key}}",
value)
    }
```

```
    return fileContents
}
```

Agora, vamos alterar a classe `TravelRequestAPITestIT` para que ela extraia o ID do passageiro e popule a requisição para a API de requisição de viagens com ele. Para isso, na chamada que é feita para a API de passageiros, precisamos chamar os métodos `extract()`, para inicializar a extração de dados da resposta; `body()`, para informar que os dados extraídos estão no corpo da resposta; `jsonPath()`, para dizer que vamos extrair usando JSON Path e `getString`, para recuperar o conteúdo e realizar a adaptação, se necessária, para a String. Esse último método recebe como parâmetro uma String JSON Path e retorna o valor desejado. Assim, a chamada para a API de passageiros fica assim:

```
val passengerId = given()
    .contentType(ContentType.JSON)

.body(loadFileContents("/requests/passengers_api/create_new_passenger.json
"))
    .post("/passengers")
    .then()
    .statusCode(200)
    .body("id", notNullValue())
    .body("name", equalToHamcrest("Alexandre Saudate"))
    .extract()
    .body()
    .jsonPath().getString("id")
```

O próximo passo é criar um mapa e populá-lo com a variável retornada e a String `passengerId` (ou seja, nosso *placeholder*):

```
val data = mapOf<String, String>(
    "passengerId" to passengerId
)
```

Agora, vamos adaptar a chamada para a API de requisições de viagens para utilizar esses *placeholders*, passando o mapa `data` como parâmetro para o método `loadFileContents`:

```
given()
    .contentType(ContentType.JSON)

.body(loadFileContents("/requests/travel_requests_api/create_new_request.j
son", data))
    .post("/travelRequests")
    .then()
    .statusCode(200)
    .body("id", notNullValue())
    .body("origin", equalToHamcrest("Avenida Paulista, 1000"))
    .body("destination", equalToHamcrest("Avenida Ipiranga, 100"))
    .body("status", equalToHamcrest("CREATED"))
    .body("_links.passenger.title", equalToHamcrest("Alexandre Saudate"))
```

Vamos também aproveitar que estamos fazendo essas melhorias e reforçar o teste da recuperação das viagens passando para a verificação o ID da requisição de viagem criada. Vamos fazer a extração desse ID de forma bastante semelhante ao que fizemos na extração do ID do passageiro, exceto o último método. Vamos passar a utilizar o método `getInt()`, que realiza *cast* para o tipo do dado JSON (no caso, um Int):

```
val travelRequestId = given()
    .contentType(ContentType.JSON)

.body(loadFileContents("/requests/travel_requests_api/create_new_request.j
son", data))
    .post("/travelRequests")
    .then()
    .statusCode(200)
    .body("id", notNullValue())
    .body("origin", equalToHamcrest("Avenida Paulista, 1000"))
    .body("destination", equalToHamcrest("Avenida Ipiranga, 100"))
    .body("status", equalToHamcrest("CREATED"))
    .body("_links.passenger.title", equalToHamcrest("Alexandre Saudate"))
    .extract()
    .jsonPath().getInt("id")
```

Finalmente, vamos alterar a última chamada para a API a fim de comparar o ID da viagem recuperada com o ID da requisição recém-

criada:

```
given()
    .get("/travelRequests/nearby?currentAddress=Avenida Paulista, 900")
    .then()
    .statusCode(200)
    .body("[0].id", equalToHamcrest(travelRequestId))
    .body("[0].origin", equalToHamcrest("Avenida Paulista, 1000"))
    .body("[0].destination", equalToHamcrest("Avenida Ipiranga, 100"))
    .body("[0].status", equalToHamcrest("CREATED"))
```

O código finalizado da nossa classe de testes fica assim:

```
package app.car.cap05.interfaces.incoming

import app.car.cap05.infrastructure.loadFileContents
import com.github.tomakehurst.wiremock.WireMockServer
import com.github.tomakehurst.wiremock.core.WireMockConfiguration
import org.junit.jupiter.api.BeforeEach
import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.boot.web.server.LocalServerPort
import org.springframework.cloud.contract.wiremock.AutoConfigureWireMock
import org.springframework.test.context.ActiveProfiles
import com.github.tomakehurst.wiremock.client.WireMock.*
import org.hamcrest.Matchers.equalTo as equalToHamcrest
import io.restassured.RestAssured
import io.restassured.RestAssured.given
import io.restassured.http.ContentType
import org.hamcrest.Matchers.notNullValue

@SpringBootTest(webEnvironment =
SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureWireMock(port = WireMockConfiguration.DYNAMIC_PORT)
@ActiveProfiles("test")
class TravelRequestAPITestIT {

    @LocalServerPort
    private var port: Int = 0
```

```
    @Autowired
    lateinit var server: WireMockServer

    @BeforeEach
    fun setup() {
        RestAssured.port = port
    }

    @Test
    fun testFindNearbyTravelRequests() {
        setupServer()
        val passengerId = given()
            .contentType(ContentType.JSON)

.body(loadFileContents("/requests/passengers_api/create_new_passenger.json
"))
            .post("/passengers")
            .then()
            .statusCode(200)
            .body("id", notNullValue())
            .body("name", equalToHamcrest("Alexandre Saudate"))
            .extract()
            .body()
            .jsonPath().getString("id")

        val data = mapOf<String, String>(
            "passengerId" to passengerId
        )

        val travelRequestId = given()
            .contentType(ContentType.JSON)

.body(loadFileContents("/requests/travel_requests_api/create_new_request.j
son", data))
            .post("/travelRequests")
            .then()
            .statusCode(200)
            .body("id", notNullValue())
            .body("origin", equalToHamcrest("Avenida Paulista, 1000"))
            .body("destination", equalToHamcrest("Avenida Ipiranga, 100"))
            .body("status", equalToHamcrest("CREATED"))
```

```
            .body("_links.passenger.title", equalToHamcrest("Alexandre
Saudate"))
            .extract()
            .jsonPath().getInt("id")

        given()
            .get("/travelRequests/nearby?currentAddress=Avenida Paulista,
900")
            .then()
            .statusCode(200)
            .body("[0].id", equalToHamcrest(travelRequestId))
            .body("[0].origin", equalToHamcrest("Avenida Paulista, 1000"))
            .body("[0].destination", equalToHamcrest("Avenida Ipiranga,
100"))
            .body("[0].status", equalToHamcrest("CREATED"))
    }

    fun setupServer() {

        server.stubFor(get(urlPathEqualTo("/maps/api/directions/json"))
            .withQueryParam("origin", equalTo("Avenida Paulista, 900"))
            .withQueryParam("destination", equalTo("Avenida Paulista,
1000"))
            .withQueryParam("key", equalTo("APIKEY"))

.willReturn(okJson(loadFileContents("/responses/gmaps/sample_response.json
")))
        )

    }
}
```

Conclusão

Neste capítulo, nós vimos como criar testes de contrato utilizando dois novos frameworks, o REST Assured e o WireMock. Enquanto um deles vai atuar na frente da nossa API, realizando requisições para ela, o outro vai atuar na retaguarda, sendo invocado pelo sistema que construímos. Ambos desempenham um papel muito

importante na criação de APIs, pois são uma garantia de qualidade e de funcionamento adequado.

Pouco a pouco, estamos caminhando rumo à construção de uma API mais robusta, semelhante a uma API do mundo real. Alguns passos interessantes podem ser vislumbrados adiante, como a adição de uma camada de segurança, por meio da qual podemos permitir, por exemplo, que apenas motoristas autenticados na plataforma sejam capazes de listar as solicitações que estão próximas.

Vamos em frente?

Capítulo 6
Segurança

"A dúvida é o princípio da sabedoria." - Aristóteles.

Com as nossas APIs já um pouco mais desenvolvidas, vamos tratar agora de aspectos relacionados à segurança. Dessa forma, garantimos que elas sejam utilizadas da maneira desejada e somente pelas pessoas a quem se destinam.

6.1 Conhecendo HTTPS

O HTTPS (`Hyper Text Transfer Protocol Secure` - protocolo de transferência de hipertexto seguro) é a peça mais fundamental de proteção de dados que temos na web. De fato, nenhum dos sistemas de autenticação e autorização que temos para nossas APIs está completo sem o uso de HTTPS. Por isso, é importante conhecermos primeiro essa peça essencial para podermos compreender com profundidade o uso dos algoritmos que vamos ver mais à frente.

Em primeiro lugar, vamos elencar três motivos pelos quais usamos HTTPS:

- **Privacidade**
- **Integridade**
- **Identificação**

Quando falamos em privacidade, quer dizer que um atacante não pode visualizar suas mensagens. Por exemplo, digamos que você está fazendo uma solicitação de viagem no aplicativo da C.A.R. para ir a um show. Esse show é de um tipo musical de *gosto duvidoso*, então você não quer que ninguém saiba que você está indo vê-lo. A

quebra de privacidade diz respeito à capacidade de um atacante de interceptar a requisição e descobrir o conteúdo.

COMO UM ATACANTE INTERCEPTA UMA MENSAGEM?

Na verdade, é surpreendentemente fácil interceptar mensagens. Existe toda uma classe de softwares chamados de *sniffers*, que são sistemas especializados em captura de pacotes através de redes. Uma vez que um atacante esteja de alguma forma conectado à rede, basta ligar o *sniffer* (por exemplo, o Wireshark - https://www.wireshark.org/) e começar a capturar e decodificar os pacotes.

O segundo ponto, integridade, quer dizer que um atacante não pode modificar sua mensagem. Então você tem a garantia de que está indo para o show, e não para um encontro de pessoas interessadas em canto mongol gutural.

O terceiro ponto, identificação, mostra que a partir da assinatura digital do servidor você tem a garantia da identidade dele. Em outras palavras, você tem a certeza de que está fazendo a requisição no aplicativo C.A.R. e não no daquela empresa que começa com U, por exemplo.

A maneira como o HTTPS trabalha previne esses três problemas através de uma camada de segurança SSL/TLS. Antes de entender como essa camada funciona, vamos falar um pouco sobre criptografia. A criptografia engloba o uso de técnicas para comunicação segura na presença de terceiros. Duas dessas técnicas mais comuns referem-se ao uso de criptografia **simétrica** ou **assimétrica**.

A **criptografia simétrica** refere-se ao uso de uma chave simétrica para o envio de mensagens. É como se você fosse enviar uma carta para um amigo e, para que ninguém inspecione o conteúdo da carta, você a coloca dentro de uma caixa trancada. Para que seu

amigo abra a caixa, ele precisa ter uma cópia da chave. Isso traz grandes problemas de gerenciamento, já que, se você tiver um grande número de amigos, é bem provável que o controle dessas chaves se torne um tanto quanto complicado - se uma delas vazar, pode comprometer as outras.

Já a **criptografia assimétrica** faz uso de duas chaves, uma **pública** e uma **privada**. A chave pública é utilizada para criptografar as mensagens, e a chave privada, para descriptografar. No exemplo da carta dentro da caixa, o processo fica mais complicado, porém mais seguro. Se você quer mandar a carta, você coloca a mensagem dentro da caixa e a tranca utilizando a **chave pública** do seu amigo. Apenas ele poderá abrir a caixa usando a **chave privada** dele.

Mas como você faz para descobrir essa chave?

A descoberta de chaves faz parte de um procedimento do HTTPS chamado de *handshake*. O *handshake* é iniciado pelo cliente, que envia o conjunto de algoritmos de criptografia (também conhecido como *cipher suite*) para o servidor que o entende. O servidor responde com seu **certificado** e a chave pública correspondente a um dos algoritmos enviados pelo cliente.

> **Um alerta!**
>
> A explicação que dou aqui sobre criptografia simétrica e assimétrica e sobre o processo de *handshake* como um todo é extremamente simplificada. Existem vários outros detalhes que achei melhor deixar de fora, por entender que não se trata do escopo deste livro. Essa é uma visão superficial contendo apenas o necessário que precisamos saber para implementar HTTPS em um serviço REST.

O **certificado** é um dos componentes principais aqui. É ele quem faz o papel de identificar o site de forma segura, através do uso de uma cadeia de confiança. Para explicar a cadeia de confiança, vou

usar uma analogia: suponha que você está em uma festa com seu amigo João. João é um dos seus melhores amigos, e você confia nele. João lhe apresenta um dos amigos dele, José. Você não conhecia José antes, mas já que João confia nele, e você confia em João, então você também confia em José. Por consequência, se José porventura lhe apresenta alguém, você também confia nessa pessoa porque você confia em José.

O funcionamento dos certificados é semelhante:

- O computador (ou qualquer outro tipo de dispositivo) vem com alguns certificados já pré-instalados nele.
- Esses certificados são das chamadas **autoridades certificadoras**, ou seja, são certificados de empresas reconhecidas pelo fabricante do sistema operacional como sendo seguras, confiáveis e capazes de realizar a verificação de que outros sites também são seguros.
- Quando entramos em um site que é acessado via HTTPS, esse site fornece um certificado que *pode* ter sido assinado por uma autoridade certificadora. Em outras palavras, essa autoridade certificadora endossa esse certificado de forma que ele também passa a ser confiável.

Dito isso, é necessário ter conhecimento de que existe também o conceito de certificado **autoassinado**, que são certificados que não são assinados por nenhuma autoridade certificadora. Esses certificados também proveem a capacidade de oferecer a privacidade e a integridade dos dados, mas são falhos na capacidade de identificação do site, pois não contam com a capacidade da cadeia de confiança. Certificados autoassinados são úteis para sistemas em fase de testes e redes internas, mas você deve evitar usá-los em APIs ou sites abertos para o mundo.

6.2 Implementando HTTPS na nossa API

Vamos criar um certificado autoassinado e implantá-lo na nossa API. A JDK já fornece uma ferramenta própria para geração de certificados, o **keytool**. Trata-se de uma ferramenta de linha de comando disponível na pasta **bin** da JDK. Na realidade, o `keytool` trabalha com arquivos conhecidos por armazenar vários objetos relacionados à criptografia, sendo os certificados apenas um desses objetos. Vamos usar o `keytool` para gerar um arquivo do tipo `PKCS12`, que é um formato interoperável para se trabalhar com várias linguagens.

Para gerar o certificado, precisamos escolher um algoritmo de criptografia. Esse algoritmo é o que vai ser usado para realizar a proteção dos dados, e o certificado vai conter a chave dele. Vários algoritmos estão disponíveis, então vamos escolher o `RSA` - um bom algoritmo de criptografia assimétrica. Por escolher esse algoritmo, também precisamos definir o tamanho da chave (quanto maior o tamanho, mais forte o algoritmo é, e maior o tempo necessário para decodificação). Vamos escolher 2048 para o tamanho (que é o padrão). Também precisamos determinar um prazo de validade para o certificado. O `keytool` recebe esse prazo em dias - vamos usar 3650 como uma aproximação para dez anos.

Os últimos parâmetros que precisamos determinar são relacionados ao `alias` do certificado (é como se fosse o nome do arquivo dentro de um arquivo `zip`), o nome do arquivo e a senha.

O comando fica assim:

```
keytool -genkeypair -alias car -keyalg RSA -keysize 2048 -storetype PKCS12
-keystore keystore.p12 -validity 3650 -storepass restbook
```

Observação: dependendo da sua instalação, pode ser necessário executá-lo com `sudo`.

Listando cada parâmetro:

- `genkeypair` determina o que esse comando está fazendo (vale lembrar que o `keytool` é uma ferramenta que tem diversas

funcionalidades diferentes).

- `alias` vai gerar o nome do certificado dentro do arquivo. No caso, o nome será `car`.
- `keyalg` é o nome do algoritmo, ou seja, `RSA`.
- `keysize` é o tamanho da chave criptográfica (2048).
- `storetype` é o tipo de arquivo que será gerado, ou seja, o `PKCS12`.
- `keystore` é o nome do arquivo físico que será gerado (`keystore.p12`).
- `validity` determina o tempo de validade do certificado (3650 dias).
- `storepass` é a senha para abertura do arquivo PKCS12 (no caso, `restbook`).

Destaco que o comando `keytool` deve ser executado na pasta `bin` da JDK. Uma vez executado, uma série de perguntas deverá ser respondida. Tenha apenas o cuidado de inserir `localhost` como resposta da primeira. As seguintes, pode responder da maneira como preferir - ou mesmo deixar em branco, como eu fiz no meu caso:

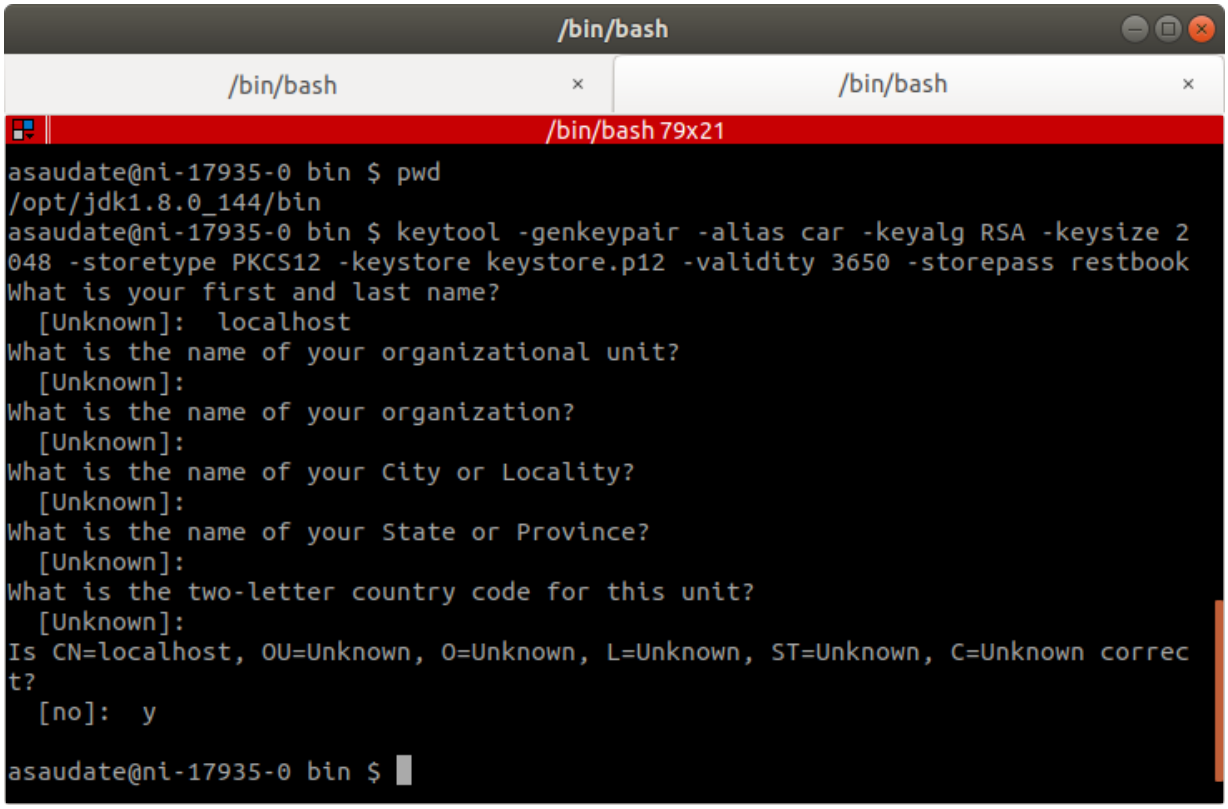

/bin/bash
/bin/bash
/bin/bash
/bin/bash 79x21
asaudate@ni-17935-0 bin $ pwd
/opt/jdk1.8.0_144/bin
asaudate@ni-17935-0 bin $ keytool -genkeypair -alias car -keyalg RSA -keysize 2
048 -storetype PKCS12 -keystore keystore.p12 -validity 3650 -storepass restbook
What is your first and last name?
[Unknown]: localhost
What is the name of your organizational unit?
[Unknown]:
What is the name of your organization?
[Unknown]:
What is the name of your City or Locality?
[Unknown]:
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]:
Is CN=localhost, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correc
t?
[no]: y
asaudate@ni-17935-0 bin $

Figura 6.1: Geração do certificado autoassinado

> **POR QUE COLOCAMOS LOCALHOST COMO CN?**
>
> O certificado serve como identificador do site que está sendo acessado, por isso é uma convenção da internet colocar o CN como o endereço onde o site está hospedado. Essa convenção deve sempre ser respeitada, pois alguns clientes podem apresentar problemas para acessar a API se o CN não corresponder ao *host*.

Agora, vamos mover o arquivo gerado para a pasta `src/main/resources` do nosso projeto. Vamos em seguida configurar o Spring Boot para habilitar o HTTPS. Para isso, precisamos colocar as seguintes configurações no arquivo `application.properties`:

```
server.ssl.key-store=classpath:keystore.p12
server.ssl.key-store-password=restbook
```

```
server.ssl.key-store-type=PKCS12
server.ssl.key-alias=car
```

A primeira configuração é relacionada à localização do arquivo `keystore.p12`. Isso indica para o Spring Boot que o arquivo está na raiz do *classpath* da aplicação. O segundo parâmetro é a senha do arquivo (que nós determinamos no parâmetro `-storepass` do `keytool`). O terceiro é o tipo do arquivo, ou seja, `PKCS12`. Finalmente, o último parâmetro é o nome do objeto que estamos procurando dentro do arquivo (que nós determinamos no parâmetro `-alias` do `keytool`).

Colocadas todas essas configurações, vamos inicializar o sistema e abrir uma API do sistema, digamos, `http://localhost:8080/drivers`. Uma mensagem semelhante à seguinte deve ser exibida:

```
Bad Request
This combination of host and port requires TLS.
```

Isso indica que o nosso servidor rejeitou a requisição por não estar com HTTPS. Vamos mudar o endereço para `https://localhost:8080/drivers` (observe que mudamos para `https` no início). Como eu uso Chrome, um alerta assim foi emitido:

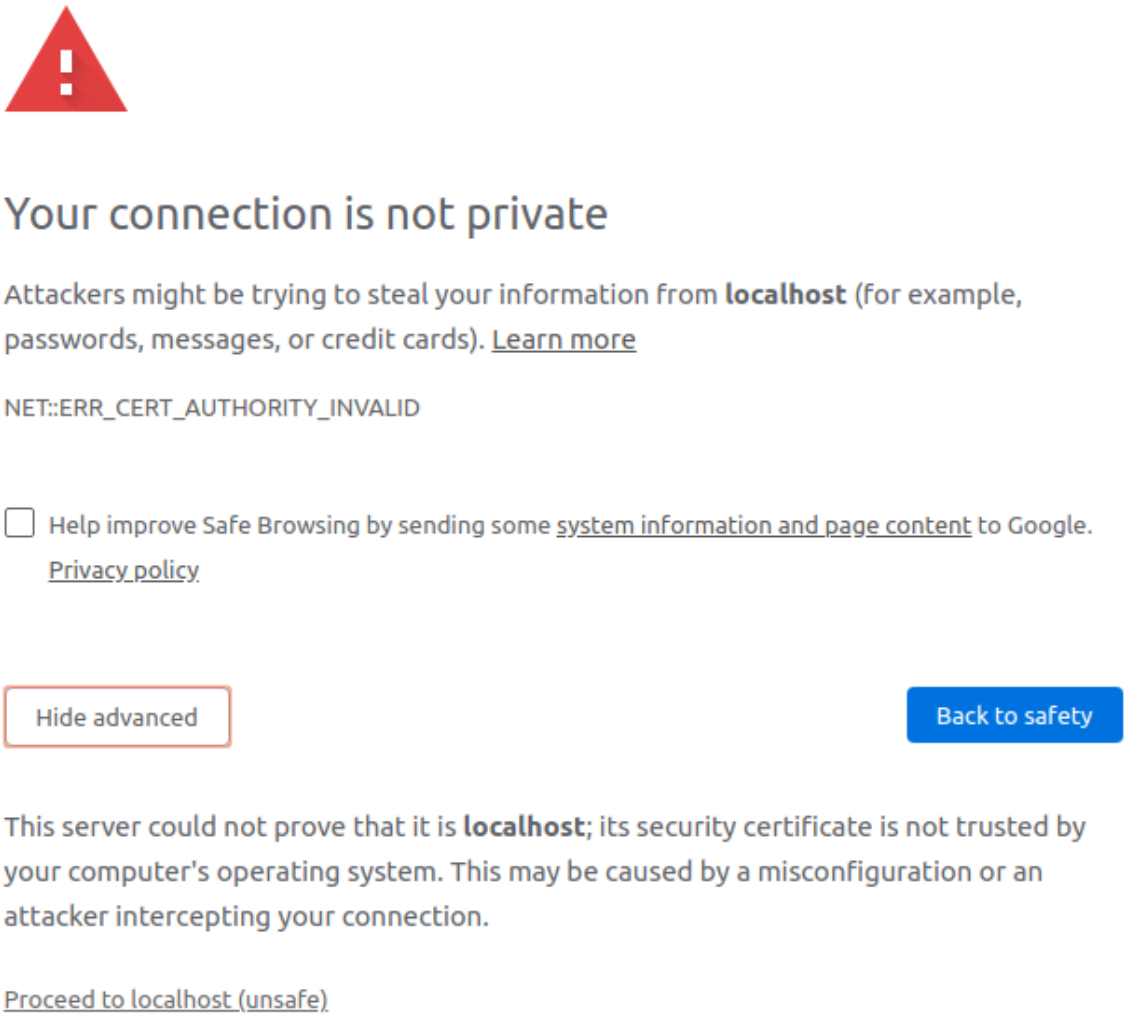

Your connection is not private
Attackers might be trying to steal your information from localhost (for example, passwords, messages, or credit cards). Learn more
NET::ERR_CERT_AUTHORITY_INVALID
Help improve Safe Browsing by sending some system information and page content to Google.
Privacy policy
Hide advanced
Back to safety
This server could not prove that it is localhost; its security certificate is not trusted by your computer's operating system. This may be caused by a misconfiguration or an attacker intercepting your connection.
Proceed to localhost (unsafe)

Figura 6.2: Chrome pedindo para liberar a request

Esse alerta significa que o Chrome entende que o certificado é autoassinado e isso não é prova da identidade do servidor. Clique em `Proceed to localhost` (ou equivalente se você usar outro *browser*). Observe que agora a API executa corretamente e um alerta aparece no canto superior.

Vamos agora verificar as requisições no Postman. Tomando a API de salvar passageiro, por exemplo, clique no botão `Send` para enviar a requisição como está. Observe que a API deve retornar um código

HTTP 400 e a mensagem indicando que a requisição deve passar por um canal seguro:

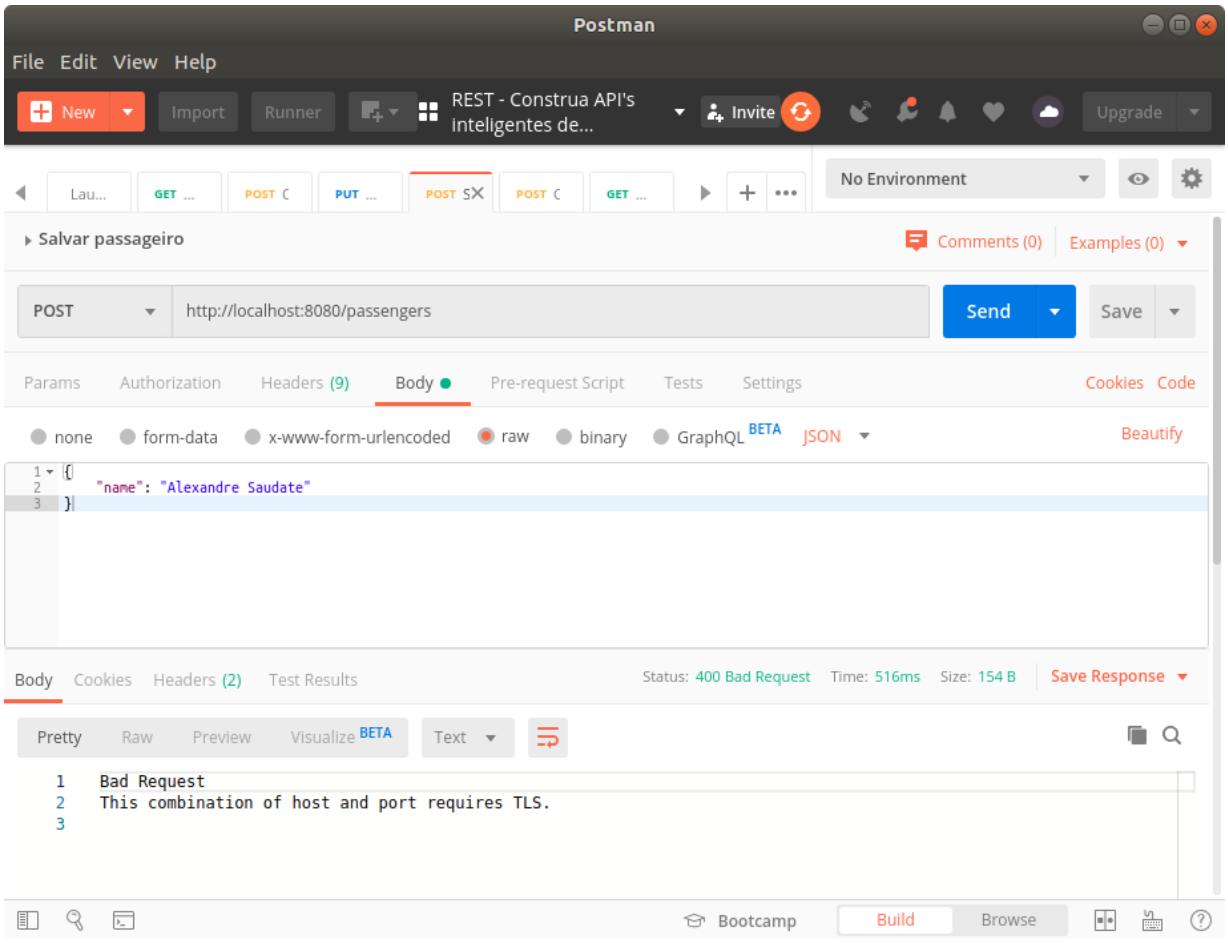


Figura 6.3: Postman apresentando a mensagem de erro do servidor

Porém, se trocarmos o host para HTTPS, o erro fica até pior:

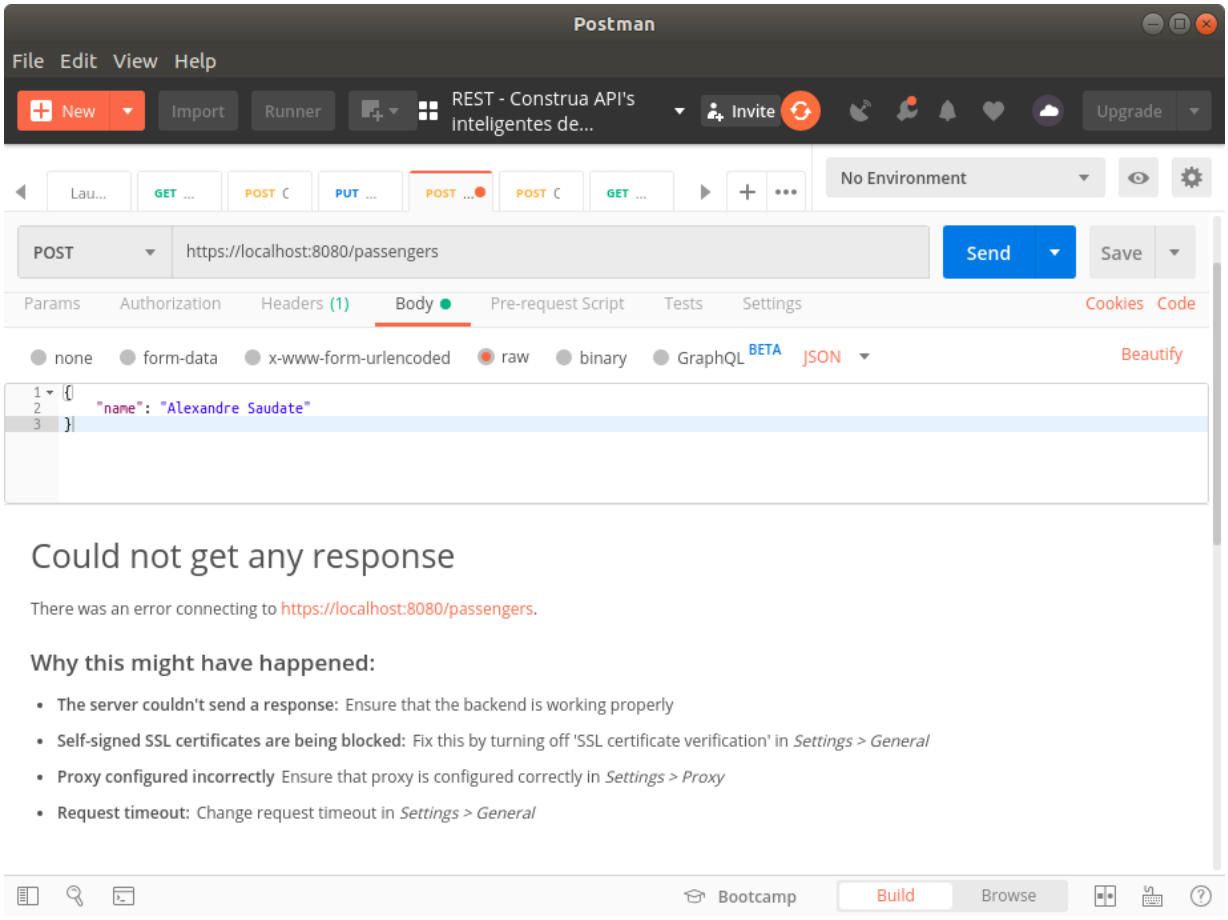


Figura 6.4: Postman mostrando uma nova mensagem de erro

Isso acontece porque o Postman valida os certificados. Como nosso certificado é autoassinado, ele se recusa a prosseguir com a requisição ainda na fase de *handshake*. Para mudar isso, clique no menu superior em `File` > `Settings` :

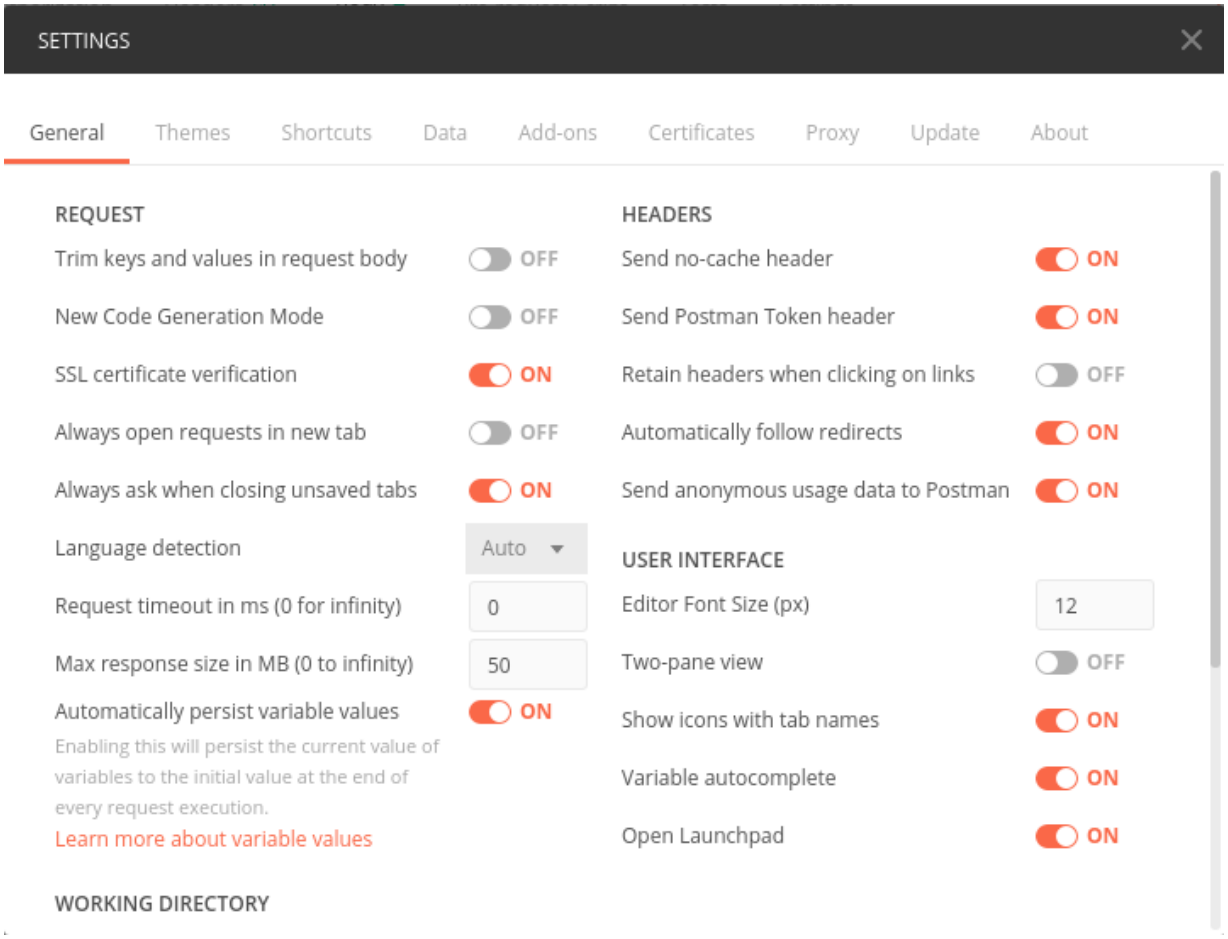


Figura 6.5: Desmarcando verificação SSL

Deixe a opção `SSL certificate validation` na posição `OFF` e teste novamente. Agora, você deve ter o resultado esperado:

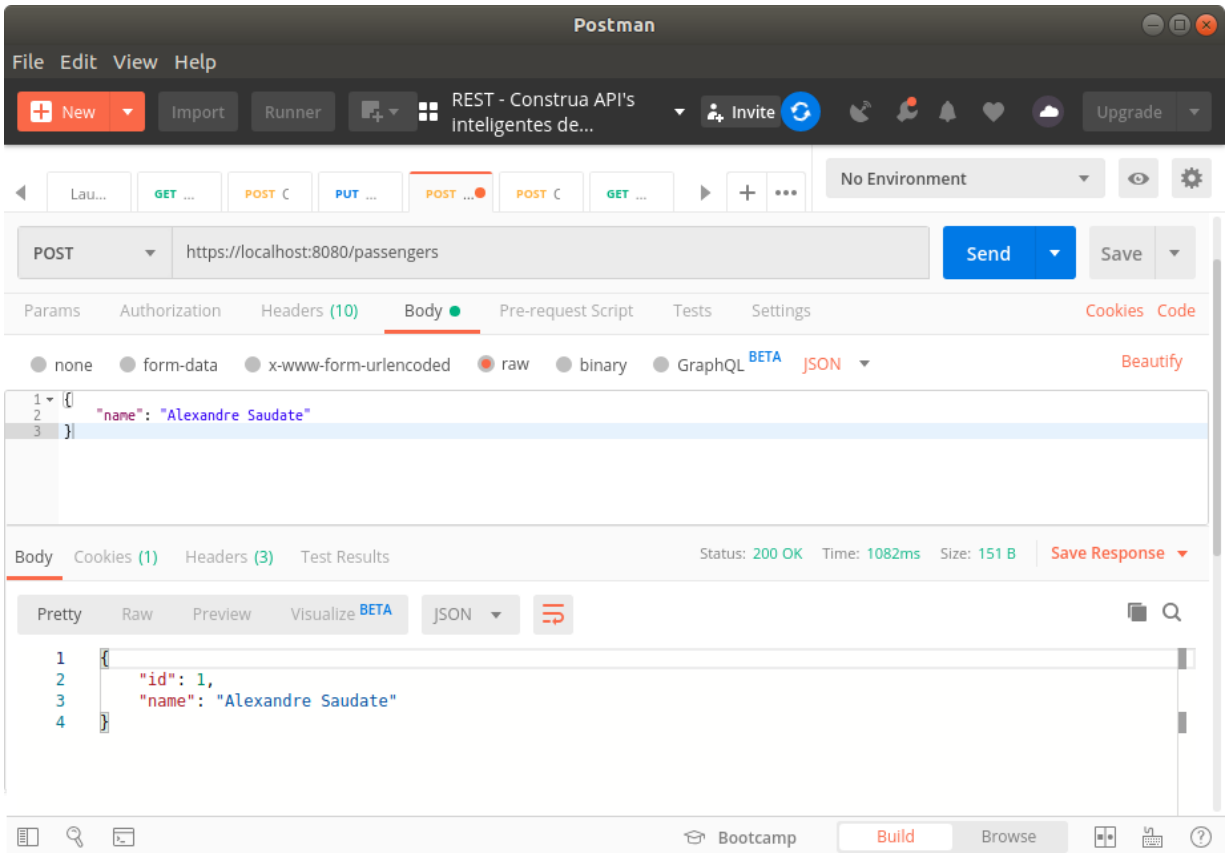


Figura 6.6: Requisição funcionando via Postman

6.3 Incluindo autenticação básica

Um dos mecanismos mais básicos da segurança da informação também é a prevenção de acesso a informações por pessoas não autorizadas. Por exemplo: na sua casa, provavelmente há uma porta. Essa porta tem uma chave. Apenas aqueles que têm uma cópia da chave podem acessar a casa, portanto a porta previne o acesso não autorizado.

O acesso a APIs (em sua forma mais simples) funciona da mesma forma. Um usuário possui uma combinação de **usuário** e **senha**, de forma que essa combinação é responsável por fazer a **autenticação** do usuário, ou seja, verificar se ele é conhecido pelo sistema, se

essa combinação é válida etc. Uma vez autenticado, é feito o processo de **autorização**, uma verificação de quais recursos o usuário pode acessar com essa combinação de usuário e senha. Voltando ao exemplo da casa, é como se abrir a porta fosse o processo de autenticação, e a autorização é como se a pessoa não pudesse acessar determinados cômodos da casa, mesmo tendo aberto a porta da frente.

Para realizar essa proteção de dados, vamos incluir um novo framework no projeto, o `Spring Security`. Para isso, basta incluir o seguinte na seção de dependências do projeto no `build.gradle.kts`:

```
implementation("org.springframework.boot:spring-boot-starter-security")
```

Note que não é necessário incluir a versão da dependência, pois o projeto já é gerenciado pelo Spring Boot e ele já faz a inclusão da versão correta, de modo que todo o conjunto opera em harmonia.

Agora, vamos criar um pacote de configurações do projeto chamado `config` e, dentro dele, um arquivo chamado `Config`:

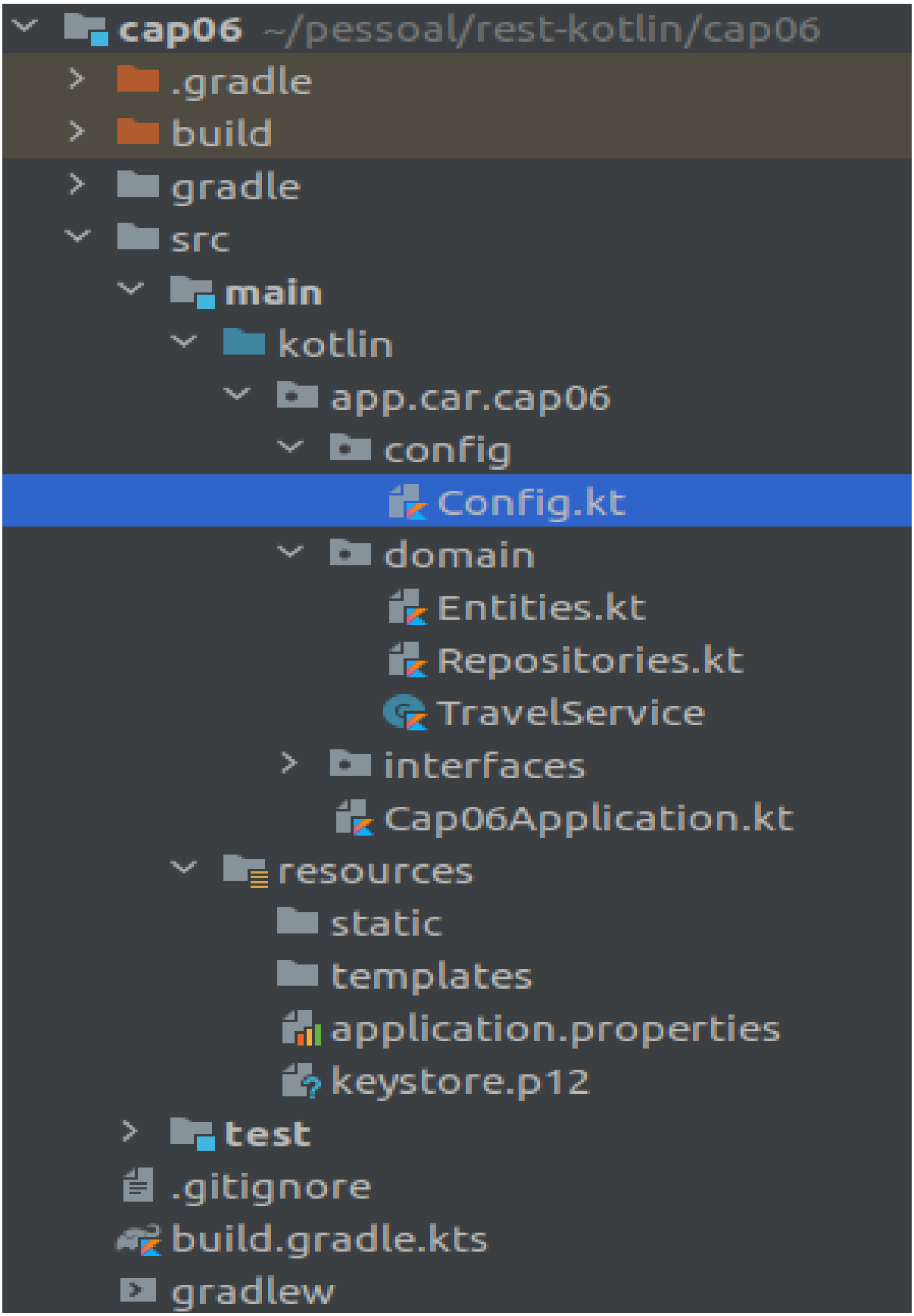

cap06 ~/pessoal/rest-kotlin/cap06
.gradle
build
gradle
src
main
kotlin
app.car.cap06
config
Config.kt
domain
Entities.kt
Repositories.kt
TravelService
interfaces
Cap06Application.kt
resources
static
templates
application.properties
keystore.p12
test
.gitignore
build.gradle.kts
gradlew

Figura 6.7: Nova estrutura do projeto

No Spring Boot, existe uma anotação chamada `@Configuration`, que determina que a classe anotada é responsável por prover objetos que estarão disponíveis para todo o contexto da aplicação. Além disso, podemos anotar essa classe com outras anotações do Spring que também vão habilitar ou desabilitar certos aspectos do sistema. Assim sendo, vamos criar uma classe chamada `SecurityConfig`, anotá-la com `@Configuration` e depois com a anotação `@EnableWebSecurity`, que vai ser responsável por habilitar efetivamente a segurança das nossas APIs:

```
package app.car.cap06.config

import org.springframework.context.annotation.Configuration
import
org.springframework.security.config.annotation.web.configuration.EnableWeb
Security

@Configuration
@EnableWebSecurity
class SecurityConfig
```

O próximo passo é configurar o nosso sistema de segurança para verificar o funcionamento. Como existem muitos aspectos a serem levados em consideração, o Spring nos fornece uma classe chamada `WebSecurityConfigurerAdapter`. O propósito dessa classe é fornecer alguns métodos para serem sobrescritos com a configuração desejada e também manter as configurações padrões para o restante. Vamos fazer com que a nossa classe `SecurityConfig` estenda `WebSecurityConfigurerAdapter`:

```
package app.car.cap06.config;

//Outros imports omitidos
import
org.springframework.security.config.annotation.web.configuration.WebSecuri
tyConfigurerAdapter
```

```
@Configuration
@EnableWebSecurity
class SecurityConfig: WebSecurityConfigurerAdapter()
```

Assim, temos acesso ao método mais importante da classe `WebSecurityConfigurerAdapter`, que é o `configure`. Esse método é sobrecarregado algumas vezes, mas o que queremos é o que recebe `HttpSecurity` como parâmetro. Vamos sobrescrever esse método para iniciar a configuração. Ele fica assim:

```
package app.car.cap06.config

//Outros imports omitidos
import
org.springframework.security.config.annotation.web.builders.HttpSecurity

@Configuration
@EnableWebSecurity
class SecurityConfig: WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {

    }
}
```

Agora vamos iniciar a configuração de segurança do nosso projeto. O primeiro passo é desabilitar a proteção a um ataque abreviado como **CSRF** (que significa *Cross-Site Request Forgery*, que traduzo como "Forja de requisições entre sites"). Primeiro vamos entender como ele funciona e por que desabilitá-lo.

A documentação do Spring detalha o CSRF com exemplos (disponível em https://docs.spring.io/spring-security/site/docs/5.4.1/reference/html5/#csrf). Vou dar aqui uma explicação resumida sobre o problema apenas para contextualizar o porquê de desabilitarmos a proteção.

Quando entramos em um site qualquer, por exemplo, um site de banco, precisamos fornecer o usuário e senha apenas uma única

vez. Como acontece, então, de o site do banco nos "reconhecer" enquanto vamos navegando por outras páginas? Através dos *cookies*. *Cookies* são pequenos arquivos de texto que são "devolvidos" para o site conforme vamos navegando naquele domínio, e através do conteúdo do *cookie* o site do banco nos reconhece, dispensando a necessidade de nos autenticarmos novamente.

Os *cookies* são fornecidos de volta apenas para os sites que os fornecem, então os *cookies* que recebemos de um site `http://abc.com.br` não são repassados para um site `http://xyz.com.br`. No entanto, o ataque de **CSRF** não precisa disso, pois um site malicioso pode incluir um formulário semelhante ao seguinte:

```
<form method="post" action="https://banco.com/transferencia">
   <input type="hidden" name="quantia" value="100.00"/>
   <input type="hidden"name="conta" value="numeroDaContaMaliciosa"/>
   <input type="submit" value="Ganhe Dinheiro Fácil, Clique Aqui Para
Saber Como!"/>
</form>
```

Isso vai fazer com que um botão escrito `Ganhe Dinheiro Fácil, Clique Aqui Para Saber Como!` apareça na tela. Quando um visitante inocente entrar no site e clicar no botão, ele fará uma requisição **legítima** para o endereço `https://banco.com/transferencia`. Por ser uma requisição legítima, o browser entende que pode fornecer os *cookies* corretos e, assim, dispensar a autenticação e a autorização necessárias. Assim, o ataque é executado com sucesso.

A proteção que o Spring Security oferece para esse ataque pode ser resumida em incluir um parâmetro extra nas requisições que é fornecido para cada requisição. O valor desse parâmetro é aleatório e gerado pelo servidor, sendo que um atacante não teria como adivinhar o valor dele. Já um visitante legítimo sim, pois a cada página acessada o servidor devolveria um novo valor para esse parâmetro, controlando a navegação.

Quando falamos de APIs REST, podemos dispensar essa proteção. Isso porque o ideal em uma API REST é que ela seja *stateless*, ou seja, sem estado. Uma API sem estado não guarda informações sobre o visitante, isto é, não usa *cookies*. Dessa forma, todo e qualquer acesso requisita os cabeçalhos de autenticação necessários, tornando as APIs automaticamente imunes ao CSRF.

Dessa forma, vamos executar dois passos: desligar a proteção contra CSRF na API, em outros termos, desativar a necessidade de fornecer esse parâmetro extra e também desligar o gerenciamento de sessões, de forma que o nosso servidor não vai mais devolver *cookies* para nenhum cliente.

Para desligar a proteção contra CSRF, invocamos o método `csrf()` e depois o método `disable()`:

```
override fun configure(http: HttpSecurity) {
    http.csrf().disable()
}
```

Para desligar o gerenciamento de sessões, invocamos o método `sessionManagement()` e depois o método `sessionCreationPolicy`, passando como parâmetro `SessionCreationPolicy.STATELESS`:

```
import org.springframework.security.config.http.SessionCreationPolicy

override fun configure(http: HttpSecurity) {
    http.csrf().disable()

    http.sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
}
```

Depois dessa preparação, vamos incluir uma autenticação básica nas nossas APIs. Essa autenticação funciona da seguinte forma:

- O primeiro passo é juntar o usuário e senha, separados por dois-pontos. Por exemplo, se o usuário for `admin` e a senha for `1234`, teremos `admin:1234`.

- Depois, passamos esse valor por um algoritmo chamado de `Base64`, que consiste na codificação de dados em um conjunto de apenas 64 caracteres (daí o nome). Nosso usuário e senha de exemplo ficariam assim: `YWRtaW46MTIzNA==`.
- O próximo passo é acrescentar a palavra `Basic` e um espaço no início. No exemplo, fica `Basic YWRtaW46MTIzNA==`.
- Por último, fornecemos esse conjunto em um cabeçalho chamado `Authorization`.

Observe que é um algoritmo muito simples e fácil de reverter para descobrir qual o usuário e senha utilizados. Por esse motivo, sempre que esse algoritmo é utilizado, é obrigatório utilizar HTTPS também.

Para configurar o sistema para utilizar esse algoritmo, utilizamos a seguinte sequência de métodos:

- `authorizeRequests()` - Para iniciar o sistema de configuração de segurança;
- `anyRequest()` - Para realizar a configuração sobre todas as URLs, com quaisquer métodos HTTP;
- `authenticated()` - Para requisitar autenticação;
- `and()` - Para complementar a configuração;
- `httpBasic()` - Para utilizar o algoritmo de autenticação básica que vimos acima.

Nosso método completo fica assim:

```
override fun configure(http: HttpSecurity) {
    http.csrf().disable()

    http.sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)

    http
        .authorizeRequests()
        .anyRequest()
        .authenticated()
        .and()
```

```
        .httpBasic()
}
```

Isso conclui a configuração do sistema para requerer os usuários. Mas e quanto aos usuários em si, onde estão? Enquanto a criação de usuários não fica pronta, o Spring Security nos fornece um usuário para realizarmos os testes. Para verificar onde está o usuário e senha, basta inicializar o sistema para visualizar algo semelhante ao seguinte:

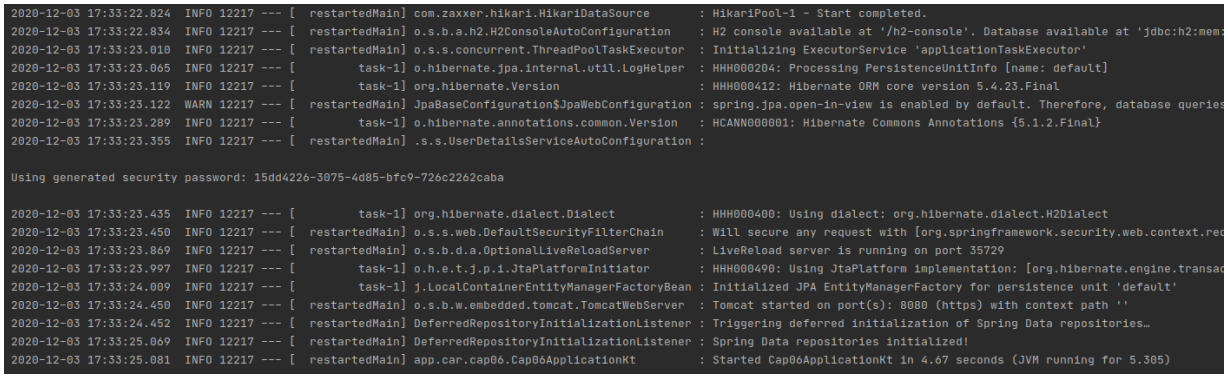

2020-12-03 17:33:22.824 INFO 12217 --- [restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2020-12-03 17:33:22.834 INFO 12217 --- [restartedMain] o.s.b.a.h2.H2ConsoleAutoConfiguration : H2 console available at '/h2-console'. Database available at 'jdbc:h2:mem
2020-12-03 17:33:23.010 INFO 12217 --- [restartedMain] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-12-03 17:33:23.065 INFO 12217 --- [task-1] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name: default]
2020-12-03 17:33:23.119 INFO 12217 --- [task-1] org.hibernate.Version : HHH000412: Hibernate ORM core version 5.4.23.Final
2020-12-03 17:33:23.122 WARN 12217 --- [restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries
2020-12-03 17:33:23.289 INFO 12217 --- [task-1] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
2020-12-03 17:33:23.355 INFO 12217 --- [restartedMain] .s.s.UserDetailsServiceAutoConfiguration :
Using generated security password: 15dd4226-3075-4d85-bfc9-726c2262caba
2020-12-03 17:33:23.435 INFO 12217 --- [task-1] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
2020-12-03 17:33:23.450 INFO 12217 --- [restartedMain] o.s.s.web.DefaultSecurityFilterChain : Will secure any request with [org.springframework.security.web.context.req
2020-12-03 17:33:23.869 INFO 12217 --- [restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2020-12-03 17:33:23.997 INFO 12217 --- [task-1] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transac
2020-12-03 17:33:24.009 INFO 12217 --- [task-1] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2020-12-03 17:33:24.450 INFO 12217 --- [restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (https) with context path ''
2020-12-03 17:33:24.452 INFO 12217 --- [restartedMain] DeferredRepositoryInitializationListener : Triggering deferred initialization of Spring Data repositories…
2020-12-03 17:33:25.069 INFO 12217 --- [restartedMain] DeferredRepositoryInitializationListener : Spring Data repositories initialized!
2020-12-03 17:33:25.081 INFO 12217 --- [restartedMain] app.car.cap06.Cap06ApplicationKt : Started Cap06ApplicationKt in 4.67 seconds (JVM running for 5.305)

Figura 6.8: Senha gerada pelo Spring Security

Observe o trecho que diz:

```
Using generated security password: 15dd4226-3075-4d85-bfc9-726c2262caba
```

Essa é uma senha gerada aleatoriamente toda vez que o sistema é inicializado. Ela é associada a um usuário chamado `user`, e podemos então passá-la para testar nossas APIs. Vamos verificar como está a configuração e testar a nossa API de criação de passageiros da forma como está. Algo semelhante ao seguinte deve ser retornado:

```
{
    "timestamp": "2019-12-26T19:06:07.724+0000",
    "status": 401,
    "error": "Unauthorized",
    "message": "Unauthorized",
    "path": "/passengers"
}
```

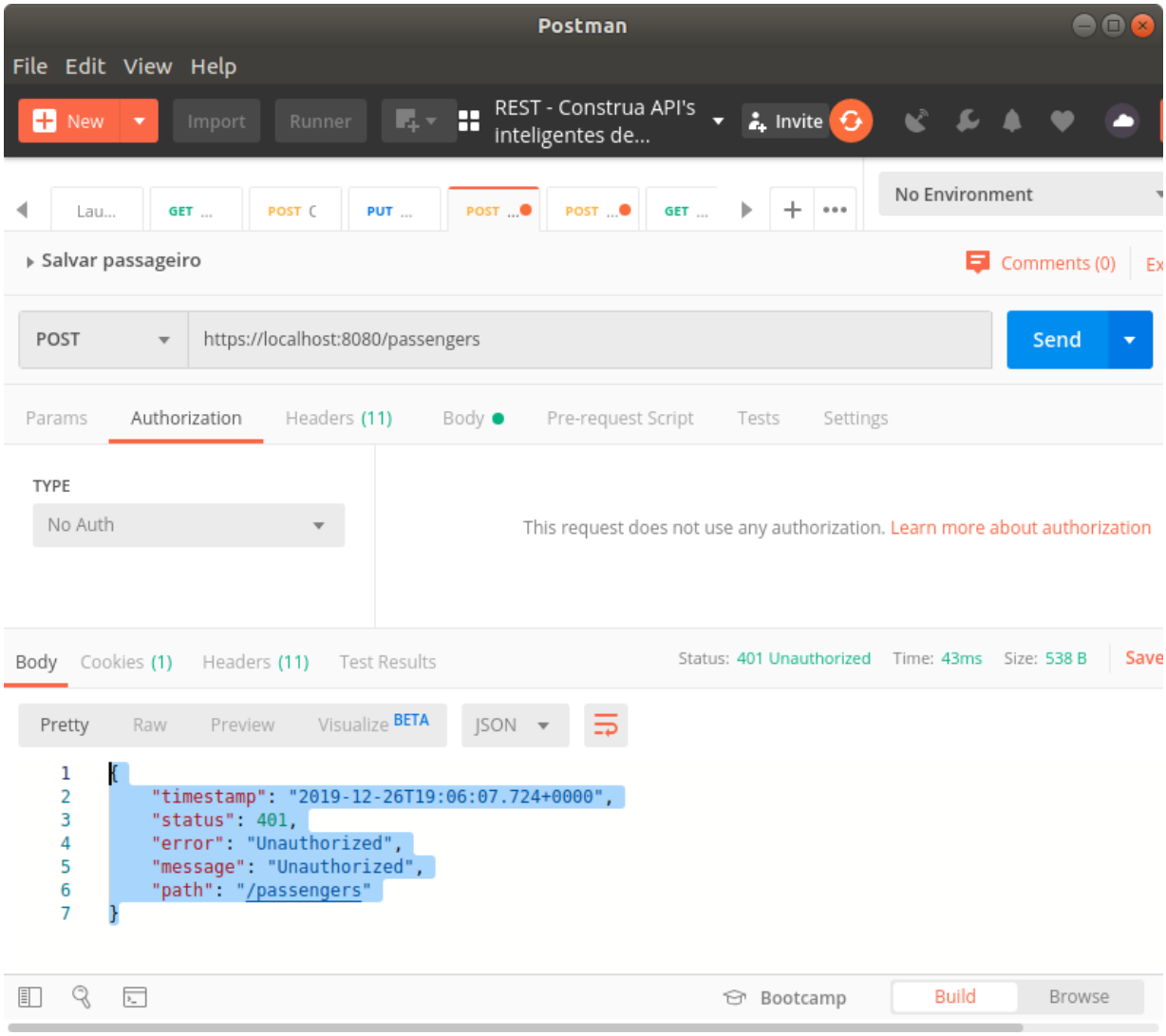


Figura 6.9: Chamada feita sem autenticação

Observe que é retornado o código de status `401`. Sua presença indica que é necessário refazer a requisição passando as credenciais corretas. Para incluir as credenciais, temos que ir até a aba `Authorization` do Postman. No *dropdown* `Type` existem vários algoritmos de autenticação disponíveis; vamos escolher `Basic Auth`. Então, vamos inserir `user` no campo `Username` e `a524c462-f10d-49f4-8bb1-398e9ede3937` (ou o equivalente no seu caso) no campo `Password`. Ao apertar `Send`, recebemos o resultado adequado:

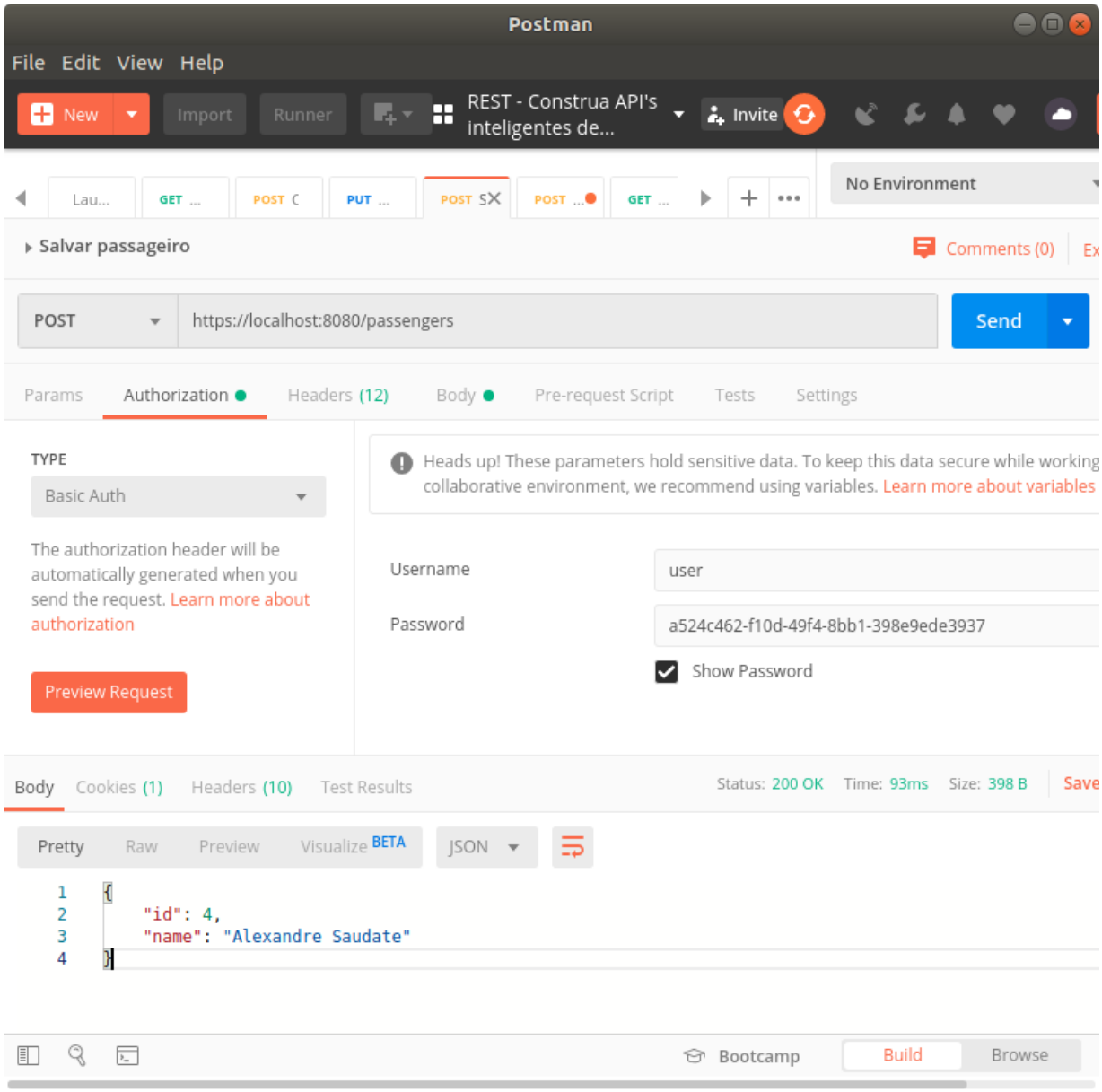


Figura 6.10: Chamada feita com autenticação

6.4 Criando sistema de autorização

O próximo passo na proteção do sistema é determinar *quem* pode acessar *o quê*. Isso quer dizer que nem todos os usuários poderão acessar as mesmas APIs. Para criarmos alguns contextos de

segurança: impedir motoristas de criar novos passageiros na plataforma, impedir motoristas de criar novas requisições de viagens na plataforma etc. Assim sendo, temos um conceito fundamental para a área de segurança da informação, o de **papéis**.

Os papéis têm a função de atribuir funções específicas para os usuários, de forma a agrupá-los sob essas funções e ficar mais fácil de determinar o que eles podem acessar. Por exemplo, fica difícil dizer que o usuário `João` não pode criar uma requisição de viagem, pois dessa forma teríamos que cadastrar restrições para todos os outros usuários, o que pode ser impraticável. Mas se `João` tiver o papel `DRIVER`, fica fácil, pois podemos simplesmente delimitar que todos os que têm esse papel não podem criar requisições de viagens.

Isso dito, vamos ver nesta seção como criar um sistema de delimitação por papéis simples, e evoluir aos poucos.

Para começar, vamos incluir a anotação `EnableGlobalMethodSecurity` na nossa classe `SecurityConfig`. Essa anotação tem a função de habilitar alguns modos de segurança por anotações. Vamos utilizar o modo de compatibilidade com a JSR-250, que é uma especificação de um conjunto de anotações para padronizar alguns aspectos da linguagem Java. Esse modo de compatibilidade é acionado a partir do atributo `jsr250Enabled`:

```
package app.car.cap06.config

//Restante dos imports omitido
import
org.springframework.security.config.annotation.method.configuration.Enable
GlobalMethodSecurity

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(jsr250Enabled = true)
class SecurityConfig: WebSecurityConfigurerAdapter() {
}
```

Com a inclusão dessa anotação, é possível utilizar a anotação `@RolesAllowed` nos métodos cujo acesso queremos restringir. Por exemplo, digamos que vamos restringir o acesso da criação de passageiros apenas àqueles que têm o papel `ADMIN`. Vamos até o método `createPassenger` na classe `PassengerAPI`, e então inserimos essa anotação e o nome do papel:

```
import javax.annotation.security.RolesAllowed;

@PostMapping
@RolesAllowed("ROLE_ADMIN")
fun createPassenger(@RequestBody passenger: Passenger)
    = passengerRepository.save(passenger)
```

O próximo passo é configurar o Spring Security para carregar os usuários e seus respectivos papéis. Para fazer isso, precisamos ir até a classe `SecurityConfig` e sobrescrever o método `configure` - desta vez, o que recebe um `AuthenticationManagerBuilder` como parâmetro:

```
import
org.springframework.security.config.annotation.authentication.builders.Aut
henticationManagerBuilder

// Trecho omitido

override fun configure(auth: AuthenticationManagerBuilder) {

}
```

Com o objeto do parâmetro `auth`, é possível definir várias formas diferentes de armazenar os usuários. No momento, para manter a simplicidade, vamos criar alguns usuários apenas em memória - posteriormente, vamos trocar essa forma de armazenamento.

Para realizar a definição dos usuários, vamos utilizar a classe `User` do Spring Security. Essa classe provê um método `builder`, que por sua vez dá acesso a vários métodos que facilitam a criação dos usuários. Em resumo, precisamos de três informações para cada

um deles: nome de usuário, senha e papéis. Para fins de comodidade, vamos criar todos com a mesma senha. Para criar a senha sem nenhuma criptografia (já que ela estará na memória do sistema, e não em nenhuma base de dados), precisamos acrescentar o prefixo `{noop}`. Esse prefixo é utilizado pelo Spring Security para saber qual sistema de codificação foi aplicado - no nosso caso, nenhum:

```
val password = "{noop}password";
```

Para realizar a definição dos usuários, vamos precisar chamar os métodos `username`, `password` e `roles` para determinar o nome de usuário, senha e papéis, respectivamente. Observe que, ao determinar os papéis, não colocamos o prefixo `ROLE_` - ele é incluído automaticamente pelo Spring Security. Assim, vamos criar três usuários, um chamado `driver` e com um papel `DRIVER`; um chamado `passenger` e com um papel `PASSENGER`; e um último chamado `admin` com um papel `ADMIN` (eu sei, muito criativo de minha parte):

```
import org.springframework.security.core.userdetails.User

//Trecho omitido

override fun configure(auth: AuthenticationManagerBuilder) {
    val password = "{noop}password";

    val driver = User.builder()
        .username("driver")
        .password(password)
        .roles("DRIVER")

    val passenger = User.builder()
        .username("passenger")
        .password(password)
        .roles("PASSENGER")

    val admin = User.builder()
        .username("admin")
        .password(password)
```

```
        .roles("ADMIN")
}
```

Agora, basta chamarmos o método `inMemoryAuthentication` e, depois, o método `withUser`, passando cada uma das definições dos usuários como parâmetro. O método completo fica assim:

```
override fun configure(auth: AuthenticationManagerBuilder) {
    val password = "{noop}password";

    val driver = User.builder()
        .username("driver")
        .password(password)
        .roles("DRIVER")

    val passenger = User.builder()
        .username("passenger")
        .password(password)
        .roles("PASSENGER")

    val admin = User.builder()
        .username("admin")
        .password(password)
        .roles("ADMIN")

    auth.inMemoryAuthentication()
        .withUser(driver)
        .withUser(passenger)
        .withUser(admin)
}
```

Vamos fazer o teste. Utilizando o Postman, vamos utilizar o usuário `driver` para tentar criar um novo passageiro:

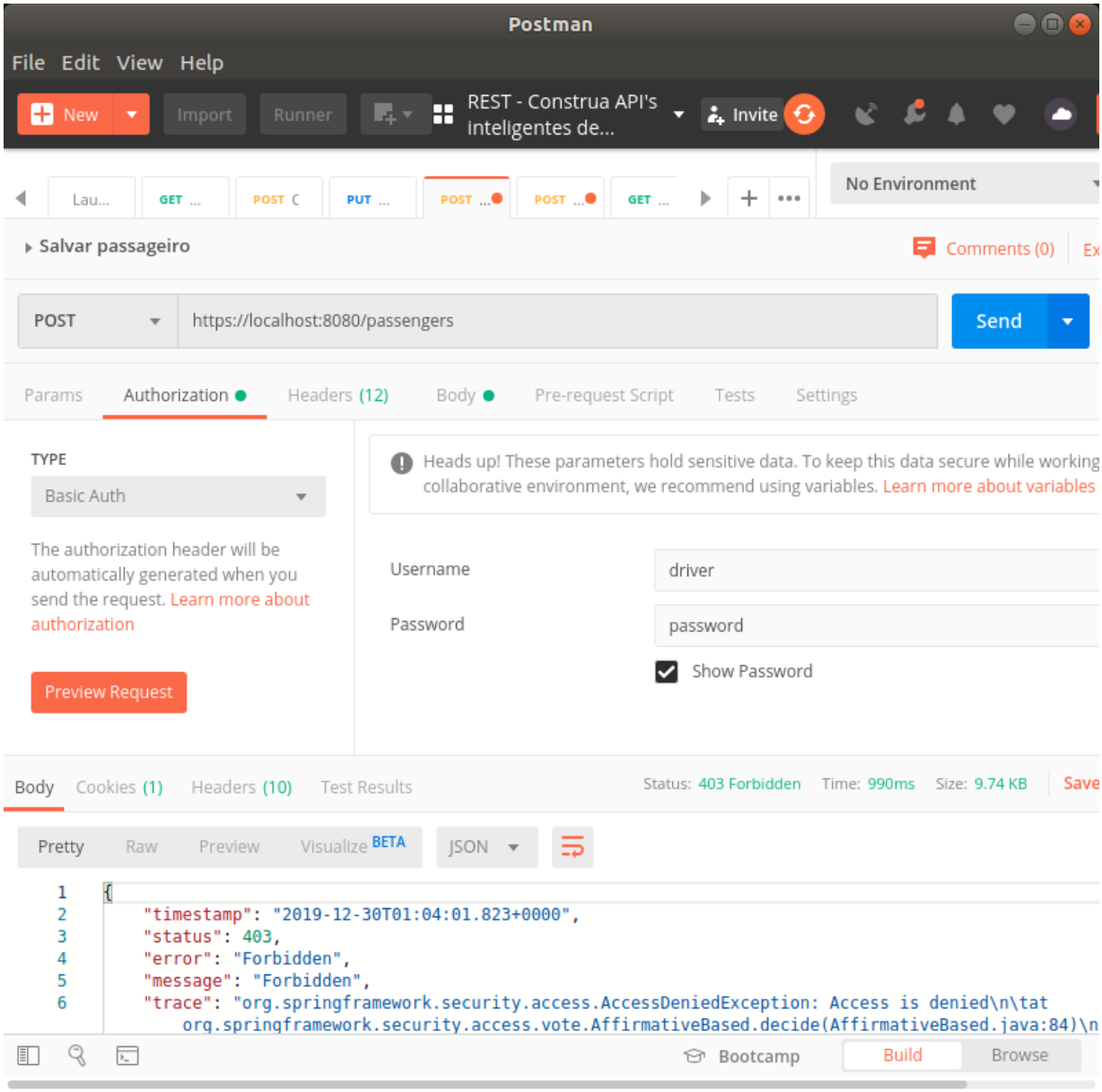


Figura 6.11: API recusando a chamada feita com usuário não autorizado

Como você pode ver, a API retorna um código `403 Forbidden` quando fazemos isso. Isso significa que o usuário foi *autenticado*, mas não *autorizado*. Em outras palavras, ele foi reconhecido como um usuário válido no âmbito do sistema, mas sem permissão para utilizar esse recurso específico do sistema.

Vamos trocar para o usuário `admin` e ver o que acontece:

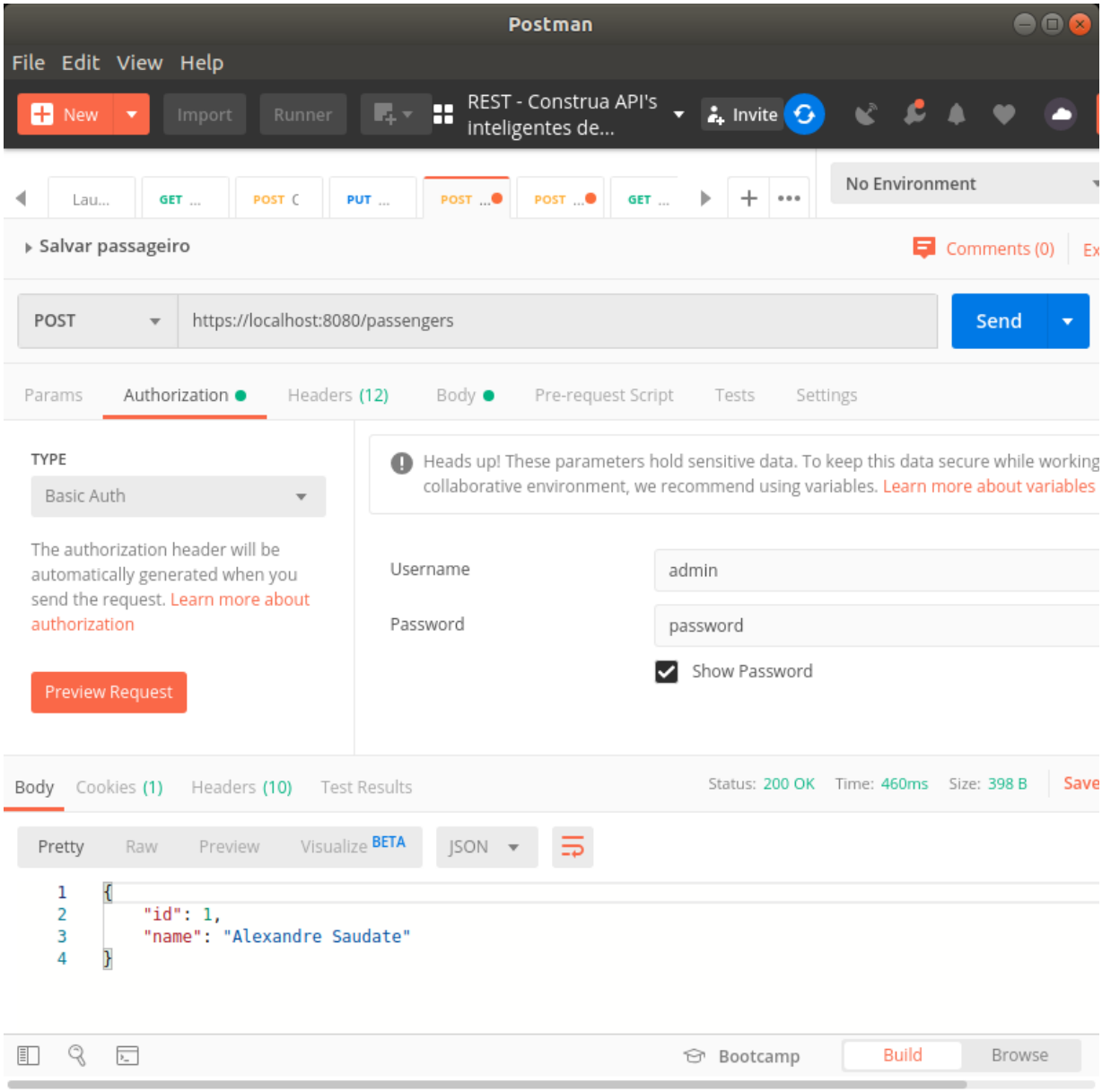


Figura 6.12: É permitido que o admin acesse o recurso

6.5 Carregando os usuários pelo banco de dados

Vamos agora iniciar a transição para um modelo no qual os usuários estejam no nosso banco de dados. Isso, claro, traz uma série de benefícios:

- Os usuários serão persistentes entre desligar e ligar novamente a aplicação (no nosso caso, isso não acontece, pois nosso banco de dados está em memória, mas isso é facilmente modificável).
- Várias instâncias diferentes do nosso programa podem enxergar o mesmo banco de dados e, portanto, os mesmos usuários.
- É possível criar novos usuários e modificá-los, e transmitir essa mudança para várias instâncias diferentes do nosso programa.

Para isso, vamos começar criando uma entidade que represente nossos usuários. Cada um deles vai contar com os seguintes atributos: um nome de usuário, uma senha, uma lista de papéis e um atributo para dizer se eles estão **habilitados** ou não. Este último é utilizado para que tenhamos a capacidade de "desligar" um usuário sem apagá-lo fisicamente do banco de dados.

Assim sendo, nossa entidade de usuário pode ficar assim:

```
package app.car.cap06.domain;

// Outros imports omitidos
import javax.persistence.Column;
import javax.persistence.ElementCollection;

@Entity
data class User(
    @Id
    @GeneratedValue
    var id: Long? = null,

    @Column(unique = true)
    val username: String,
    val password: String,
    val enabled: Boolean = true,
```

```
    @ElementCollection
    val roles: MutableList<String>

)
```

O QUE É A ANOTAÇÃO @ELEMENTCOLLECTION?

Essa anotação é utilizada para que possamos realizar o mapeamento de classes simples para o banco de dados. Da forma como está, isso vai fazer com que uma outra tabela seja criada, chamada `user_roles`. Essa tabela vai conter uma coluna chamada `user_id`, que vai ser mapeada para o `id` do usuário, e uma coluna chamada `roles`. Esta coluna vai conter os valores da lista. O JPA já faz todo esse mapeamento de forma transparente para nós.

Vamos configurar o Spring Security para carregar esses usuários do banco de dados. Para fazer isso, precisamos de duas *queries*, uma para carregar os usuários propriamente ditos e outra para carregar os papéis associados a esses usuários. Essas *queries* estarão diretamente vinculadas às tabelas que serão criadas com base na classe `User` que foi descrita anteriormente.

A *query* para carregar o usuário precisa retornar o nome de usuário, a senha, e o atributo que indica se está habilitado ou não. Além disso, essa *query* recebe como parâmetro o nome de usuário fornecido, que pode ser representado na *query* por um sinal de interrogação (`?`). Assim, a *query* fica:

```
select username, password, enabled from user where username=?
```

Já a *query* que lista os papéis é ligeiramente mais complexa, pois vai utilizar a tabela que foi gerada com base na anotação `@ElementCollection`. Essa *query* precisa retornar o nome de usuário e os papéis, ou seja, ela pode retornar várias linhas para um único usuário. Para fazer isso, precisaremos correlacionar as duas tabelas

criadas, `user` e `user_roles`, utilizando o campo `id` da tabela `user` e `user_id` da tabela `user_roles`. Essa *query* também recebe como parâmetro o `username`.

Com tudo isso, vamos utilizar a *query* assim:

```
select u.username, r.roles from user_roles r, user u where r.user_id =
u.id and u.username=?
```

Com as *queries* já criadas, vamos modificar a classe `SecurityConfig` para que ela possa realizar a carga dos usuários a partir do banco. Para isso, em primeiro lugar vamos injetar um `datasource` na classe `SecurityConfig`. Este `datasource` é uma abstração para representar uma fonte de dados, ou seja, um banco de dados:

```
import javax.sql.DataSource

//Restante do código omitido

class SecurityConfig(
    val datasource: DataSource
): WebSecurityConfigurerAdapter() {

   //Restante do código omitido

}
```

Vamos também começar a cuidar de um aspecto muito importante da segurança da nossa aplicação: o armazenamento de senhas. Para toda aplicação que armazena senhas de qualquer tipo, uma regra muito importante é armazená-las de modo que, caso o banco de dados seja comprometido, não seja possível (ou seja muito difícil) reverter o que está armazenado de volta para as senhas verdadeiras. Como via de regra, armazenamos `hashes` das senhas em vez da senha original. Um `hash` é um código gerado a partir da senha original de forma que uma mesma senha gere uma saída previsível, mas que o processo seja irreversível. Para determinar se uma senha é equivalente ao `hash` dela, o sistema tira o `hash` da

senha fornecida pelo cliente e verifica se é equivalente ao `hash` armazenado. Se for igual, o processo de autenticação é feito com sucesso.

> **SE O HASH SEMPRE GERA A MESMA SAÍDA, NÃO BASTA TER UM DICIONÁRIO DE SAÍDAS PARA REVERTER O PROCESSO?**
>
> É válido pensar que, se os usuários quase sempre usarem a senha `123456`, por exemplo, teremos armazenado no banco de dados vários `hashes` iguais e, dessa forma, saber que a senha do usuário é `123456`, mesmo enxergando apenas o `hash`. Com base nisso, foi criado o conceito de `salt`, que é um dado conhecido pelo gerador de `hashes` e que é anexado na senha que vai passar pelo processo. Dessa forma, o `salt` faz com que o `hash` da mesma senha seja diferente para diferentes usuários, o que dificulta bastante o processo de adivinhar qual a senha de um determinado usuário.
>
> O sistema utilizado pelo Spring Security por padrão (que vamos ver a seguir) utiliza o `salt` nativamente, portanto não vamos precisar nos preocupar com esse aspecto da segurança.

Para utilizar esse sistema, o Spring Security nos fornece a interface `PasswordEncoder`. Basta termos uma implementação dessa interface disponível no contexto do Spring que ele já a utiliza automaticamente em diversos contextos, mas não em todos. Assim, vamos criar um método que insere a implementação `BCryptPasswordEncoder` no contexto do Spring - o que é feito criando-se um método público na classe `SecurityConfig` anotado com `@Bean`:

```
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
import org.springframework.context.annotation.Bean

class SecurityConfig(
    val datasource: DataSource
): WebSecurityConfigurerAdapter() {
```

```
    @Bean
    fun passwordEncoder() = BCryptPasswordEncoder()

    // Restante do código omitido
}
```

Finalmente, estamos prontos para reconfigurar o sistema de autenticação. Para isso, vamos deixar comentado o sistema que tínhamos criado anteriormente, com os usuários em memória:

```
override fun configure(auth: AuthenticationManagerBuilder) {
    /*
    val password = "{noop}password";

    val driver = User.builder()
        .username("driver")
        .password(password)
        .roles("DRIVER")

    val passenger = User.builder()
        .username("passenger")
        .password(password)
        .roles("PASSENGER")

    val admin = User.builder()
        .username("admin")
        .password(password)
        .roles("ADMIN")

    auth.inMemoryAuthentication()
        .withUser(driver)
        .withUser(passenger)
        .withUser(admin)

    */
}
```

Vamos colocar as *queries* que tínhamos criado anteriormente em Strings:

```
val queryUsers = "select username, password, enabled from user where
username=?"
val queryRoles = "select u.username, r.roles from user_roles r, user u
where r.user_id = u.id and u.username=?"
```

E agora, vamos modificar o objeto `auth` para buscar os dados no banco de dados utilizando o método `jdbcAuthentication`. Para que ele saiba em qual banco de dados buscar as informações, vamos injetar a instância do `datasource` utilizando o método `dataSource()`:

```
auth.jdbcAuthentication().dataSource(datasource)
```

Vamos fornecer uma instância do `PasswordEncoder` para que não haja problemas com a autenticação:

```
auth.jdbcAuthentication()
    .dataSource(datasource)
    .passwordEncoder(passwordEncoder())
```

Por último, vamos fornecer as *queries* para recuperar os usuários e os papéis através dos métodos `usersByUsernameQuery` e `authoritiesByUsernameQuery`, respectivamente:

```
auth.jdbcAuthentication()
    .dataSource(datasource)
    .passwordEncoder(passwordEncoder())
    .usersByUsernameQuery(queryUsers)
    .authoritiesByUsernameQuery(queryRoles)
```

Para conferência, a classe completa ficou assim:

```
package app.car.cap06.config

import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import
org.springframework.security.config.annotation.authentication.builders.Aut
henticationManagerBuilder
import
org.springframework.security.config.annotation.method.configuration.Enable
GlobalMethodSecurity
```

```
import
org.springframework.security.config.annotation.web.builders.HttpSecurity
import
org.springframework.security.config.annotation.web.configuration.EnableWeb
Security
import
org.springframework.security.config.annotation.web.configuration.WebSecuri
tyConfigurerAdapter
import org.springframework.security.config.http.SessionCreationPolicy
import org.springframework.security.core.userdetails.User
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder
import javax.sql.DataSource

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(jsr250Enabled = true)
class SecurityConfig(
    val datasource: DataSource
) : WebSecurityConfigurerAdapter() {

    @Bean
    fun passwordEncoder() = BCryptPasswordEncoder()

    override fun configure(http: HttpSecurity) {
        http.csrf().disable()

        http.sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)

        http
            .authorizeRequests()
            .anyRequest()
            .authenticated()
            .and()
            .httpBasic()
    }

    override fun configure(auth: AuthenticationManagerBuilder) {
        /*
        val password = "{noop}password";
```

```
        val driver = User.builder()
            .username("driver")
            .password(password)
            .roles("DRIVER")

        val passenger = User.builder()
            .username("passenger")
            .password(password)
            .roles("PASSENGER")

        val admin = User.builder()
            .username("admin")
            .password(password)
            .roles("ADMIN")

        auth.inMemoryAuthentication()
            .withUser(driver)
            .withUser(passenger)
            .withUser(admin)

         */

        val queryUsers = "select username, password, enabled from user
where username=?"
        val queryRoles = "select u.username, r.roles from user_roles r,
user u where r.user_id = u.id and u.username=?"

        auth.jdbcAuthentication()
            .dataSource(datasource)
            .passwordEncoder(passwordEncoder())
            .usersByUsernameQuery(queryUsers)
            .authoritiesByUsernameQuery(queryRoles)
    }
}
```

Para testar, vamos criar um modo de salvar nosso usuário no banco de dados e nos autenticar com ele. Para isso, primeiro vamos criar um repositório de usuários, da mesma forma como criamos os das outras entidades do sistema:

```
interface UserRepository: JpaRepository<User, Long>
```

Agora, vamos criar uma nova configuração para que nossa aplicação possa recuperar os objetos adequados e realizar a persistência de um `User` no banco de dados. Vamos criar uma nova classe chamada `LoadUserConfig` (dentro do arquivo `Config.kt`) e anotá-la com `@Configuration`:

```
@Configuration
class LoadUserConfig {
}
```

A seguir, vamos injetar tanto o `PasswordEncoder` quanto o `UserRepository`:

```
import org.springframework.security.crypto.password.PasswordEncoder
import app.car.cap06.domain.UserRepository

// Restante dos imports omitidos

@Configuration
class LoadUserConfig(
    val passwordEncoder: PasswordEncoder,
    val userRepository: UserRepository
) {
}
```

Vamos agora criar um método que será responsável por executar a criação do usuário. Podemos sinalizar para o Spring que esse método deverá ser invocado após a instanciação da classe, utilizando a anotação `@PostConstruct`:

```
import javax.annotation.PostConstruct

@Configuration
class LoadUserConfig(
    val passwordEncoder: PasswordEncoder,
    val userRepository: UserRepository
) {

    @PostConstruct
    fun init() {
```

```
    }
}
```

Finalmente, vamos codificar o método `init` para que um usuário seja inserido no repositório e possamos fazer nossos testes. Para manter um contexto adequado com o que estávamos trabalhando antes, este usuário vai ser o `admin`, com a senha `password`, um único papel `ROLE_ADMIN` e, obviamente, habilitado. A senha vai precisar estar codificada, conforme explanado anteriormente. Assim, vamos utilizar o método `encode` antes de passar a senha para o usuário. A criação toda fica:

```
import app.car.cap06.domain.User as DomainUser

@PostConstruct
fun init() {
    val admin = DomainUser(
        username = "admin",
        password = passwordEncoder.encode("password"),
        roles = mutableListOf("ROLE_ADMIN")
    )
    userRepository.save(admin)
}
```

Dessa forma, assim que iniciarmos o sistema, vamos ter um usuário `admin` criado no banco de dados. Vamos testar novamente usando o Postman. Basta deixar o usuário criado como `admin` e a senha como `password`, e realizar a requisição para criação do passageiro:

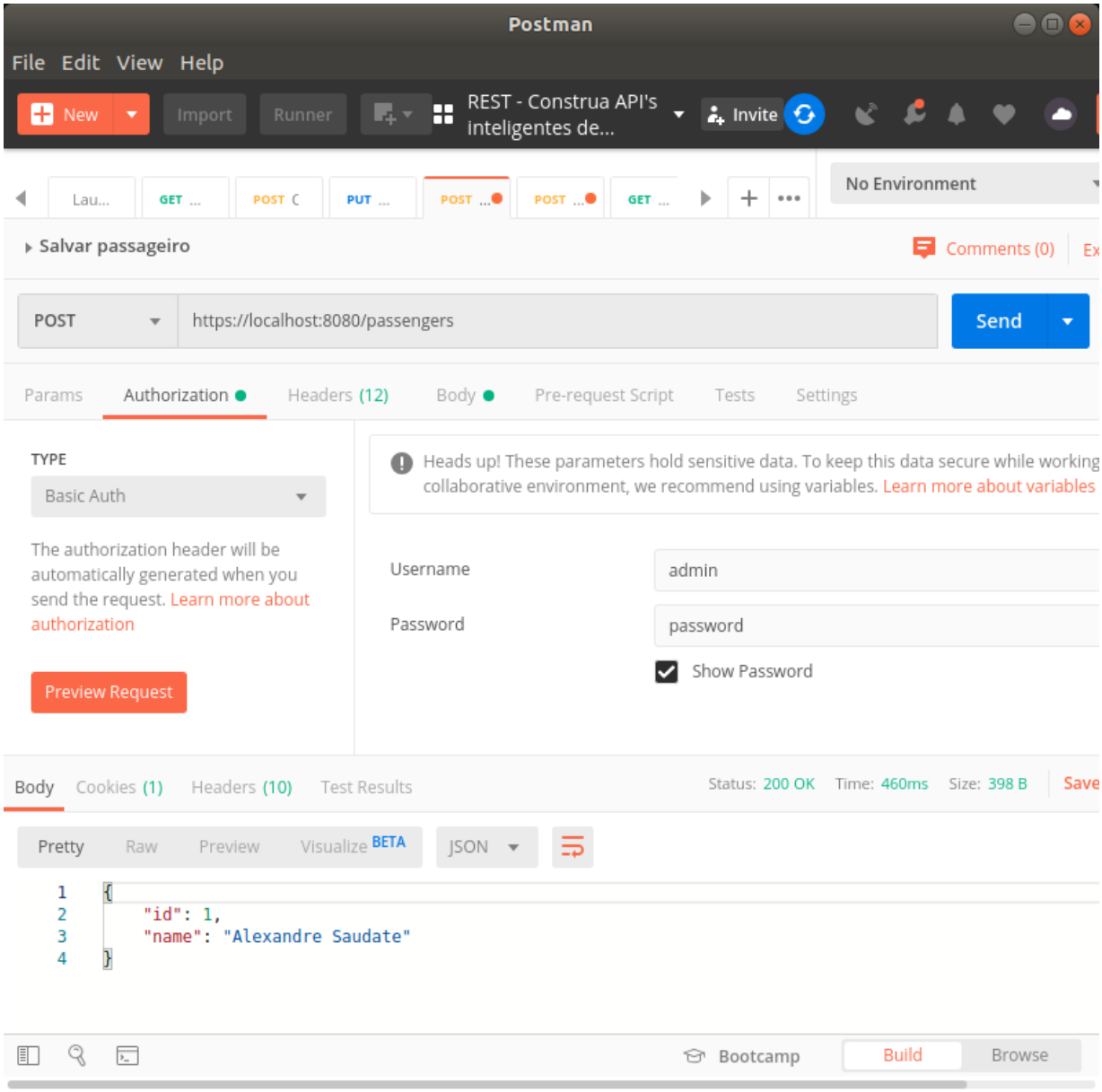


Figura 6.13: Passageiro criado com o usuário admin

6.6 Atualização dos testes integrados

Como atualizamos o nosso sistema para utilizar HTTPS e autenticação, os testes com o REST Assured precisam refletir essas novas configurações. Primeiro vamos alterar os testes para que o

REST Assured utilize HTTPS em vez do padrão HTTP. Isso pode ser feito utilizando o atributo estático `baseURI` da classe, que define para o REST Assured qual será a URI base de todas as requests efetuadas por ele. Se esse atributo for configurado para um endereço com HTTPS, ele sempre utilizará HTTPS nas requests feitas.

Tomando como base a classe `PassengerAPITestIT`, vamos modificar o método `setup()` para que utilize esse atributo. Observe que, uma vez que utilizamos esse atributo, temos que especificar a porta utilizada - dessa forma, dispensando o uso do atributo `port` utilizado anteriormente:

```
@BeforeEach
fun setup() {
    RestAssured.baseURI = "https://localhost:$port"
}
```

Ao executar o código dessa forma, podemos nos deparar com uma mensagem semelhante à seguinte: `javax.net.ssl.SSLHandshakeException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target`. Isso acontece por causa do certificado que é autoassinado (semelhante ao problema que ocorria com o Postman). Para contornar esse problema, utilizamos o método `useRelaxedHTTPSValidation()` do REST Assured:

```
@BeforeEach
fun setup() {
    RestAssured.baseURI = "https://localhost:$port"
    RestAssured.useRelaxedHTTPSValidation()
}
```

Quando executamos novamente o teste, temos o seguinte erro:

```
java.lang.AssertionError: 1 expectation failed.
Expected status code <200> but was <401>.
```

Como vimos anteriormente, esse código de status indica que não estamos utilizando autorização. Podemos passar a utilizá-la através

do atributo estático `authentication` do REST Assured. Para indicar que é um sistema de autenticação Basic, vamos utilizar o método estático `basic()` para informar as credenciais do nosso usuário `admin`:

```
import io.restassured.RestAssured.basic

@BeforeEach
fun setup() {
    RestAssured.baseURI = "https://localhost:$port"
    RestAssured.useRelaxedHTTPSValidation()
    RestAssured.authentication = basic("admin","password")
}
```

Agora, os testes vão passar. Vamos replicar a mesma alteração para a classe `TravelRequestAPITestIT`, para que os testes passem.

> **ALERTA DE BOAS PRÁTICAS**
>
> Caso você tenha muitos testes, pode ser interessante utilizar uma estratégia de separar essas configurações em uma classe à parte e apenas invocar a chamada no teste. No caso dos exemplos deste livro, não considerei proveitoso fazer isso por serem apenas dois. No entanto, se você tiver curiosidade a respeito da melhor forma de como fazer isso, não se esqueça de conferir o repositório oficial de códigos deste livro em https://github.com/alesaudate/rest-kotlin.

Conclusão

Até aqui, vimos alguns dos conceitos mais básicos de segurança de APIs. Conhecemos o mecanismo de funcionamento de certificados, HTTPS e autenticação básica para usuários dentro do sistema. Do ponto de vista de serviços, pode-se dizer que os nossos estão concluídos.

Mas criar e disponibilizar APIs são mais do que isso. Trata-se de criar serviços que apresentam boa documentação e, de certa forma, "chamam" novos usuários, tamanha é a usabilidade dos serviços.

Na próxima parte, vamos conhecer técnicas para expandir a usabilidade dos serviços que implementamos criando esse convite para seus usuários.

APIs

Capítulo 7
APIs

"Todos os homens têm, por natureza, desejo de conhecer" - Aristóteles.

A partir deste capítulo, quero convidar você a vislumbrar um novo passo na evolução dos nossos serviços. Nesta parte, vamos aprender a converter os serviços que estamos construindo em REST para algo que seja verdadeiramente fácil de se utilizar, e que possa realmente instigar os usuários a *quererem* utilizar as APIs.

Vamos dissecar o assunto a partir da definição do que é uma API. Esta é a sigla para *Application Programming Interface*, ou Interface de programação de aplicações. Trata-se de uma forma de comunicação com o software que procura abstrair totalmente os detalhes da implementação subjacente - ou ao menos tanto quanto for possível.

Para entender melhor esse conceito, vamos detalhar o que é uma *interface*. Intuitivamente, você conhece interfaces, pois você já teve que lidar com algumas delas. Quando você ligou seu computador, em algum momento você viu a interface gráfica do seu sistema operacional (Windows, Linux, Mac, ou qualquer outro). Repare no termo: **interface** gráfica. Isso significa que o seu sistema operacional oferece uma abstração de natureza gráfica para que você não tenha que lidar com detalhes inerentes à implementação. Em outras palavras, o computador oferece facilidades que estão disponíveis graficamente para que você não tenha que acessar a implementação interna do sistema e mexer diretamente no código interno dele.

Além desse exemplo, vale observar um que está mais próximo da realidade dos programadores Java, isto é, as próprias interfaces que a linguagem disponibiliza para nós. Quando declaramos uma interface em Java, ela não tem implementação alguma (pelo menos, até o Java 8 não tinha). Isso forçava o programador a exercer uma técnica chamada **desenvolvimento orientado a interfaces**, ou seja, declarar métodos pensando na **usabilidade** em primeiro lugar, e abstraindo o máximo possível quaisquer detalhes sobre a implementação deles.

Entendemos que uma API é uma interface para o programador, e uma interface é uma forma de se interagir com o sistema sem conhecer seu funcionamento interno. Mas qual a importância dessas APIs?

Em primeiro lugar, temos que entender que o propósito de uma API é tornar o uso do sistema **fácil** e **conveniente** para que, por meio dele, programadores não familiarizados com esse sistema possam criar códigos rapidamente. Essa tem sido a base de muitos sistemas que conhecemos hoje, e realmente é bem difícil estabelecer um *roadmap* de sucesso de um novo sistema na internet sem uma API disponível. Quando nós temos uma API para o público utilizar, expandimos o público-alvo do nosso sistema e este passa a contemplar também desenvolvedores que queiram integrar suas funcionalidades em seus próprios sistemas.

Um grande exemplo desse cenário são as redes sociais, como o Facebook. Uma vez que ele oferece algumas de suas funcionalidades expostas por APIs, ele mesmo expande seu público-alvo para desenvolvedores que queiram construir funcionalidades utilizando o Facebook. Uma desenvolvedora pode oferecer *login* em seu site via Facebook, ou pode construir um *chatbot* para o próprio Facebook, ou pode ainda detectar algumas das preferências do usuário. Enfim, existem várias possibilidades a serem construídas a partir do uso de APIs - e isso não diminui a popularidade do próprio Facebook, pelo contrário, só aumenta. Isso porque mais e mais sistemas passam a ser, na verdade,

dependentes do Facebook para que essas funcionalidades permaneçam no ar.

Para se implementar uma boa API, é fundamental compreender que é necessário que essa API seja tão próxima quanto possível de ser autoexplicativa e também ter uma boa documentação. Isso significa seguir algumas regras, tais como:

- Ter URLs significativas.
- Utilizar os códigos HTTP corretos.
- Em caso de erros, é importante fornecer mensagens de erro detalhando os erros.
- Caso seja uma API pública, fazer com que ela seja retrocompatível.
- Ter uma boa documentação.

7.1 Como criar URLs significativas

Como vimos até aqui, cada URL que criamos estava relacionada a conceitos simples, tais como `/drivers` para motoristas, `/passengers` para passageiros, e assim por diante. Mas o que fazer em caso de conceitos aparentemente relacionados, como as viagens já relacionadas por um determinado motorista ou passageiro?

Geralmente, nesses casos, quem desenvolve costuma recorrer à hierarquização de URLs. Por exemplo, para listar as viagens já percorridas por um determinado motorista (digamos, identificado por `1`), cria-se uma URL `/drivers/1/travels`. Apesar de esse desenvolvimento parecer inofensivo à primeira vista, essa ideia pode levar à explosão de novas URLs para lidar com esses conceitos ou correlatos e, em casos mais extremos, tornar muito difícil ou impossível a manutenção da API.

O correto seria tratar de cada URL como a representação de um conjunto - no sentido mais próximo que isso pode estar do conceito

matemático.

Explico: ao desenvolver a URL `/drivers`, criamos a identificação de um conjunto no qual estão contidos diversos motoristas. Podemos selecionar alguns deles de acordo com um filtro aplicado, por exemplo, `/drivers?firstName=Alexandre`, ou podemos selecionar um motorista específico de acordo com um subconjunto da URL - `/drivers/1`. Ao optar por esta última, no entanto, note que criamos um subconjunto do conjunto inicial, e não apenas filtramos. Esse subconjunto em particular contém apenas um motorista.

Nessa linha de raciocínio, já começa a parecer mais difícil "encaixar" o conceito de viagens. Afinal de contas, precisamos ter a noção de pertencimento: o motorista 1 faz parte do conjunto de motoristas, pois ele é um motorista. Já as viagens que ele fez não são motoristas, são viagens. São conceitos **relacionados**, mas não têm uma relação de pertencimento.

Observe que aqui também se encaixam alguns conceitos clássicos de Orientação a Objetos: alguns relacionamentos podem ser do tipo TEM-UM (por exemplo: um cachorro TEM UMA coleira) e outros relacionamentos podem ser do tipo É-UM (um cachorro É UM animal).

A partir desse conceito, percebe-se por que a URL `/drivers/1/travels` está conceitualmente errada. Para estar correta, seria necessário que as viagens fossem motoristas. Mas o relacionamento entre eles é do tipo TEM-UM e não É-UM. Daí entra a necessidade de se utilizar HATEOAS. Quando conceitos estão relacionados desta forma, faz mais sentido.

Além disso, também vale ressaltar que, como as URLs são uma representação de conjuntos, não faz sentido algum nomeá-las utilizando verbos. Em geral, utilizamos apenas substantivos e, em alguns casos mais complexos, adjetivos (como foi o caso da API que criamos no capítulo 4 `/travelRequests/nearby`). Observe que, por mais que levemos em consideração a noção de conjuntos

matemáticos equivalentes a cada segmento de URL, o segmento `/nearby` é um adjetivo e, portanto, está filtrando `travelRequests`. Em outras palavras, o conjunto representado por `/travelRequests` contém o subconjunto representado por `/nearby`.

Quando você tiver dúvidas de que tipo de mecanismo utilizar para filtragens (via segmento de URL ou `query params`), prefira `query params` por ser um mecanismo mais extensível. Utilize segmentos de URL apenas quando não fizer sentido qualquer espécie de filtragem extra dentro do conjunto em questão.

7.2 Utilização dos códigos HTTP corretos

Conhecemos ao longo do livro alguns dos códigos HTTP mais usados. Veremos agora um detalhamento um pouco maior desses códigos para entender quando usar cada um deles. Primeiro, vamos entender as "famílias" dos códigos HTTP para entender a classificação deles:

1xx

Códigos que iniciam com 1 são apenas informativos e não costumam ser de grande utilidade em contextos HTTP/REST.

2xx

São os códigos de sucesso. Os mais usados são:

- `200 OK`: é uma resposta "genérica" de sucesso, e indica que no resultado da requisição deve haver um corpo.
- `201 Created`: indica que o recurso foi criado. Deve incluir um cabeçalho `Location` indicando a URL onde o recurso está disponível.
- `202 Accepted`: significa que a requisição foi recebida, mas será processada de forma assíncrona.

- `204 No Content` : a requisição foi processada, mas não há conteúdo algum para ser retornado no corpo.

3xx

São códigos de redirecionamento, ou seja, a resposta depende de outras requisições. Existe uma grande fonte de confusão nesta família (que família não tem confusões, não é mesmo?), que é o fato de que muitos *browsers* populares implementaram alguns códigos com o comportamento de outros. Devido a este fato, o comportamento de alguns códigos passa a ser muito semelhante, então vou procurar agrupar aqueles que têm comportamento similar:

- `301 Moved Permanently` : indica que o recurso presente nesta URL mudou-se permanentemente e novas requisições devem ser feitas para o novo endereço.
- `302 Found` e `303 See Other` : significa que o resultado do processamento realizado por esta URL está presente em outro endereço. Isso quer dizer que, caso o usuário tenha feito uma requisição com POST ou PUT, por exemplo, o resultado vai estar em outra URL e deve ser obtido usando GET.
- `304 Not Modified` : não é de redirecionamento. É usado em cenários onde alguma técnica de *cache* foi usada, indicando que o recurso não foi modificado no servidor e que o usuário pode reaproveitar a mesma versão que ele tem salva localmente.
- `307 Temporary Redirect` : semelhante ao `302` e `303` (veja parágrafo a seguir para mais detalhes).

A confusão que envolve os códigos `302` , `303` e `307` se dá por uma implementação errônea dos fabricantes de *browsers* e por erros nas especificações de cada um. Essa falha de entendimento já foi corrigida e, hoje, o código `302` não deve mais ser usado. Em substituição, utilize o `301` caso o redirecionamento seja permanente; `303` , caso o redirecionamento seja temporário e a nova requisição precise ser feita com o método `GET` ; e `307` , caso o mesmo método utilizado para realizar a requisição no primeiro endereço precise ser

usado para realizar a requisição no novo endereço (por exemplo, se eu utilizei POST para realizar uma requisição em `/travelRequests` e essa requisição retornou o código `307`, então eu devo realizar uma nova requisição utilizando o método `POST` no endereço apontado).

4xx

São códigos de erros que se originam a partir de alguma ação tomada por parte do cliente. Geralmente, eles podem ser corrigidos se o cliente mudar alguma informação na requisição. Incluem erros na formatação dos dados enviados, autenticação/autorização, conflito com os dados já contidos no servidor etc.

- `400 Bad Request`: é uma resposta genérica de erro causado por uma ação do cliente. Pode ser utilizada para indicar que um tipo de dado não está bem formatado ou não é aceito, por exemplo, enviar letras em um campo de CEP.
- `401 Unauthorized`: indica que a autenticação é requerida, mas os dados não foram encontrados ou falharam (senha errada, por exemplo). Curiosamente, o texto de status desse código significa "Não Autorizado" quando, na opinião deste autor, deveria ser "Não Autenticado".
- `403 Forbidden`: indica que a autorização necessária não foi concedida para acessar a URL em questão. Nesse caso, a própria especificação alerta que, caso os dados de autenticação não sejam modificados, o pedido não deve ser reenviado.
- `404 Not Found`: é um dos códigos de erro mais usados de todos. Indica que o recurso que deveria estar presente em uma determinada URL não está.
- `409 Conflict`: é utilizado para casos de conflito entre recursos. Nasceu com o propósito de ser utilizado onde mecanismos de *lock* otimista estão presentes; no entanto, por significar que existe algo em conflito no servidor, teve seu uso expandido para outros casos, como quando o próprio cliente fornece o ID do recurso a ser criado para o servidor e o referido ID já está em uso.

5xx

São códigos de erros que se originaram a partir de algo errado que aconteceu no lado do servidor. Esses erros geralmente não podem ser corrigidos a partir de correções na requisição e apenas os mantenedores do serviço podem corrigir as falhas.

- `500 Internal Server Error` : é uma resposta genérica de erro interno do servidor. Por si só, indica apenas alguma falha não tratada no servidor. Geralmente, o código que escrevemos no Spring Boot já traduz a resposta para esse código automaticamente caso qualquer exceção seja propagada.
- `503 Service Unavailable` : é uma resposta que indica que o servidor não está disponível. Esse erro geralmente é dado quando a requisição HTTP que fizemos é redirecionada internamente para que outro serviço atenda à requisição e este serviço não está disponível.

Vale lembrar que essa lista não está completa (longe disso), por dois motivos principais: o primeiro é que nem todos os códigos definidos pela especificação HTTP são utilizáveis em REST. Temos até o caso de um código que é uma pegadinha de primeiro de abril e acabou entrando para a lista oficial de códigos de status (se você tiver curiosidade, é o código `418 I'm a teapot` , que indica que o servidor não pode preparar café por ser um bule de chá). O segundo motivo é que muitos códigos têm contextos muito específicos e acabam sendo muito pouco usados (um exemplo é o código `450 Blocked by Windows Parental Controls` , o que significa que a requisição foi bloqueada pelo controle parental do Windows).

7.3 Fornecer mensagens de erro significativas

Uma questão importante a ser trabalhada quando falamos de APIs é prover uma boa experiência relacionada aos casos de erro. Esses

erros devem ser claros, de modo a orientar, da melhor forma possível, o cliente a tratá-los (caso sejam erros da família `4xx`) ou como obter algum *feedback* (caso sejam erros da família `5xx`).

Veremos como realizar esse tipo de tratamento a partir da validação de dados, utilizando a API de solicitação de viagens como exemplo. Vamos revisar a classe `TravelRequestInput`:

```
data class TravelRequestInput(
    val passengerId: Long,
    val origin: String,
    val destination: String
)
```

Essa classe fica no pacote que delimita entradas das APIs do sistema, então é importante que ela tenha algumas validações de informações para minimizar a possibilidade de entrada de dados que provoquem erros no sistema. Vamos examinar uma possível entrada de erros: o método `map` da classe `TravelRequestMapper`. Esse método recebe como parâmetro um `TravelRequestInput` e faz a transformação deste para um `TravelRequest`, que é um objeto de domínio. Vamos revisar o código:

```
fun map(input: TravelRequestInput) : TravelRequest {

    val passenger = passengerRepository.findById(input.passengerId!!)
        .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }

    return TravelRequest(passenger = passenger,
        origin = input.origin!!,
        destination = input.destination!!)
}
```

A primeira coisa que esse método faz é localizar os dados do passageiro a partir do `passengerId` passado como parâmetro. Mas e se esse atributo não for fornecido na requisição?

Como fizemos antes, vamos utilizar o Postman para verificar essa situação realizando um POST na URL `/travelRequests`. Vamos

enviar os seguintes dados:

```
{
  "origin": "string",
  "destination": "string"
}
```

Observe que o `passengerId` foi omitido. Da forma como está, recebemos uma resposta semelhante à seguinte:

```
{
    "timestamp": "2020-12-03T23:09:23.363+00:00",
    "status": 404,
    "error": "Not Found",
    "trace": "org.springframework.web.server.ResponseStatusException: 404
NOT_FOUND\n\tat

app.car.cap07.interfaces.incoming.mapping.TravelRequestMapper$map$passenge
r$1.get(TravelRequestMapper.kt:27)\n\tat

app.car.cap07.interfaces.incoming.mapping.TravelRequestMapper$map$passenge
r$1.get(TravelRequestMapper.kt:18)\n\tat

java.base/java.util.Optional.orElseThrow(Optional.java:408)\n\tat

app.car.cap07.interfaces.incoming.mapping.TravelRequestMapper.map(TravelRe
questMapper.kt:27)\n\tat

app.car.cap07.interfaces.incoming.TravelRequestAPI.makeTravelRequest(Trave
lRequestAPI.kt:32)\n\tat

java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native
Method)\n\tat

java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMetho
dAccessorImpl.java:62)\n\tat

java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(Delegat
ingMethodAccessorImpl.java:43)\n\tat

java.base/java.lang.reflect.Method.invoke(Method.java:566)\n\tat
```

```
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(Inv
ocableHandlerMethod.java:197)
    "path": "/travelRequests"
}
```

Recebemos um erro HTTP 404, indicando que o passageiro não foi encontrado. Isso acontece porque o Kotlin identifica que o parâmetro é um tipo primitivo e não pode ser nulo. Nesse cenário, atribui-se um valor padrão para o campo - no caso, 0. Como não existe um passageiro com o ID 0 no banco de dados, o erro é lançado. Esse código é adequado para um passageiro que não foi encontrado, mas o que queremos de fato é que seja retornada uma exceção indicando que o ID do passageiro não foi fornecido. Ou seja, o retorno correto deveria ser um HTTP 400. Vamos iniciar o tratamento para isso.

O primeiro passo é tornar os campos da classe `TravelRequestInput` nulos pelo Kotlin. Isso porque é mais fácil detectar se o campo foi ou não fornecido dessa forma. Então, vamos colocar um ponto de interrogação nos tipos dos campos:

```
data class TravelRequestInput(
    val passengerId: Long?,
    val origin: String?,
    val destination: String?
)
```

Isso vai fazer com que a compilação quebre nos pontos onde esses campos são usados, portanto, no método `map` da classe `TravelRequestMapper`. Para resolver o problema, vamos utilizar dois pontos de exclamação nas chamadas, que indicam para o Kotlin que o campo não será nulo ali:

```
fun map(input: TravelRequestInput) : TravelRequest {

    val passenger = passengerRepository.findById(input.passengerId!!)
        .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }
```

```
    return TravelRequest(passenger = passenger,
        origin = input.origin!!,
        destination = input.destination!!)
}
```

O próximo passo é introduzir a devida validação nos campos para que o sistema do Spring Boot não deixe a requisição prosseguir no código caso alguma informação esteja incorreta (no nosso caso, qualquer um dos campos na requisição faltando). Isso pode ser feito através do uso das anotações referentes à JSR 380, também conhecida como `Bean Validation 2.0`. Essa especificação inclui anotações para validação de vários aspectos do sistema (inclusive dados válidos apenas no Brasil, como CPF e CNPJ), e o Spring Boot já se integra automaticamente a ela.

Para acrescentar o sistema de validação no projeto, inclua a seguinte declaração no `build.gradle.kts`:

```
implementation("org.springframework.boot:spring-boot-starter-validation")
```

Vamos utilizar a anotação `@NotNull` para declarar que o campo `passengerId` não pode ser nulo - sendo assim, as requisições que vierem sem esse campo deverão ser recusadas. Para que o sistema funcione, o sistema da validação precisa que a anotação esteja presente no *getter* da propriedade. Neste caso, vamos colocar o prefixo `get:` na anotação:

```
import javax.validation.constraints.NotNull

data class TravelRequestInput(
    @get:NotNull
    val passengerId: Long?,

    val origin: String?,
    val destination: String?
)
```

Agora, no ponto da chamada que declara o método POST, vamos incluir a anotação `@Valid`, que indica que o `Bean Validation` deve ser

utilizado sobre este parâmetro:

```
import javax.validation.Valid;

//Restante do código omitido

@PostMapping
fun makeTravelRequest(@RequestBody @Valid travelRequestInput:
TravelRequestInput)
    : EntityModel<TravelRequestOutput> {
    val travelRequest =
travelService.saveTravelRequest(mapper.map(travelRequestInput))
    val output = mapper.map(travelRequest)
    return mapper.buildOutputModel(travelRequest, output)
}
```

Se realizarmos a requisição do Postman agora, já vamos ter um resultado bem diferente:

```
{
    "timestamp": "2020-02-16T19:03:14.142+0000",
    "status": 400,
    "error": "Bad Request",
    "errors": [
        {
            "codes": [
                "NotNull.travelRequestInput.passengerId",
                "NotNull.passengerId",
                "NotNull.java.lang.Long",
                "NotNull"
            ],
            "arguments": [
                {
                    "codes": [
                        "travelRequestInput.passengerId",
                        "passengerId"
                    ],
                    "arguments": null,
                    "defaultMessage": "passengerId",
                    "code": "passengerId"
```

```
                }
            ],
            "defaultMessage": "??? null",
            "objectName": "travelRequestInput",
            "field": "passengerId",
            "rejectedValue": null,
            "bindingFailure": false,
            "code": "NotNull"
        }
    ],
    "message": "Validation failed for object='travelRequestInput'. Error
count: 1",
    "trace":
"org.springframework.web.bind.MethodArgumentNotValidException: Validation
failed for argument [0] in public
org.springframework.hateoas.EntityModel<app.car.cap07.interfaces.incoming.
output.TravelRequestOutput>
app.car.cap07.interfaces.incoming.TravelRequestAPI.makeTravelRequest(app.c
ar.cap07.interfaces.incoming.input.TravelRequestInput): [Field error in
object 'travelRequestInput' on field 'passengerId': rejected value [null];
codes
[NotNull.travelRequestInput.passengerId,NotNull.passengerId,NotNull.java.l
ang.Long,NotNull]; arguments
[org.springframework.context.support.DefaultMessageSourceResolvable: codes
[travelRequestInput.passengerId,passengerId]; arguments []; default
message [passengerId]]; default message [must not be null]] \n\tat
org.springframework.web.servlet.mvc.method.annotation.RequestResponseBodyM
ethodProcessor.resolveArgument(RequestResponseBodyMethodProcessor.java:139
)\n\tat
org.springframework.web.method.support.HandlerMethodArgumentResolverCompos
ite.resolveArgument(HandlerMethodArgumentResolverComposite.java:121)\n\tat
org.springframework.web.method.support.InvocableHandlerMethod.getMethodArg
umentValues(InvocableHandlerMethod.java:167)\n",
    "path": "/travelRequests"
}
```


Já está muito melhor: o código HTTP foi 400 (que era nossa intenção), e existe uma mensagem indicando que houve algo de errado relacionado ao campo `passengerId`. Mas ainda não está

amigável o suficiente e, pior de tudo, não seria possível utilizar a mesma estrutura de erro para outros cenários.

Dessa forma, precisamos criar uma estrutura na qual seja possível definir um *payload* que seja extensível para todos os novos cenários de falha. Para o nosso caso, em específico, um JSON simples nos atenderia muito bem. Por exemplo:

```
{
   "errors": [
      {
         "message":"O campo passengerId não pode ser nulo"
      }
   ]
}
```

Perceba que se trata de uma estrutura externa (que é onde o campo `errors` está) e uma estrutura interna (que abriga o campo `message`). Vamos então criar um pacote chamado `app.car.cap07.interfaces.incoming.errorhandling` para armazená-las e um arquivo contendo duas classes que representam essas duas estruturas, começando pela mais interna. Esta classe contém dados específicos sobre cada erro, então vamos chamá-la de `ErrorData`:

```
package app.car.cap07.interfaces.incoming.errorhandling

data class ErrorData(
    val message: String
)
```

Em seguida, vamos criar a segunda classe, mais externa, que conterá uma lista de objetos `ErrorData`. Por ser algo mais global, vamos chamá-la de `ErrorResponse`:

```
data class ErrorResponse(
    val errors: List<ErrorData>
)
```

Agora, vamos criar um sistema de captura da falha e "tradução" para as classes que acabamos de criar. Para isso, o Spring MVC

disponibiliza duas anotações que têm o propósito de facilitar esse processo. A primeira é a `@RestControllerAdvice`, que, como o nome indica, intercepta as invocações dos métodos das APIs. A segunda é a `@ExceptionHandler`, que é parametrizada com uma exceção que pode eventualmente ser lançada pelo método da API (por exemplo, a exceção que é lançada caso a validação da entrada falhe).

Quando criamos esse tratador, é necessário que seu retorno seja mapeável pelo Spring como retorno da API. Isso porque, uma vez tratada a exceção, o retorno desse método é o que será utilizado como retorno da API. Assim sendo, podemos criar um método que retorne o nosso `ErrorResponse`.

O último ponto que falta definir é qual exceção precisamos tratar, tanto para parametrizar a anotação `@ExceptionHandler` quanto para parametrizar o método que vamos criar. Se você observar bem, verá que o nome da exceção que precisamos tratar já apareceu: ela foi lançada assim que inserimos o tratamento para o campo faltante e que fez com que o código HTTP `400` fosse lançado. Esta exceção é a `MethodArgumentNotValidException`.

Já temos o suficiente para realizar a declaração do método tratador:

```
package app.car.cap07.interfaces.incoming.errorhandling

import org.springframework.web.bind.MethodArgumentNotValidException
import org.springframework.web.bind.annotation.ExceptionHandler
import org.springframework.web.bind.annotation.RestControllerAdvice

@RestControllerAdvice
class DefaultErrorHandler {

    @ExceptionHandler(MethodArgumentNotValidException::class)
    fun handleMethodArgumentNotValid(ex: MethodArgumentNotValidException):
ErrorResponse {
        return ErrorResponse(emptyList())
    }
}
```

Os dados de que precisamos para realizar o tratamento do erro estão todos contidos na exceção que, convenientemente, recebemos como parâmetro do método. Esta exceção contém um objeto do tipo `BindingResult`, que, por sua vez, contém uma lista de objetos do tipo `FieldError`. Esses `fieldErrors` contêm uma série de dados a respeito do que originou a falha - no momento, vamos nos ater ao campo `defaultMessage`, que é a mensagem de erro gerada pelo sistema para indicar a falha.

Como nós temos um construtor na classe `ErrorData` que recebe uma String como parâmetro, podemos utilizar a função `map` para traduzir a lista de `FieldError` em uma lista de `ErrorData`:

```
@ExceptionHandler(MethodArgumentNotValidException::class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
fun handleMethodArgumentNotValid(ex: MethodArgumentNotValidException):
ErrorResponse {
    val messages = ex.bindingResult.fieldErrors.map {
        ErrorData(it.defaultMessage!!)
    }
    return ErrorResponse(messages)
}
```

O último passo que precisamos ajustar é o código HTTP que será retornado quando essa exceção for tratada. Mais uma vez, podemos utilizar uma anotação do próprio Spring para indicar qual é o código, a `@ResponseStatus`. Essa anotação é parametrizada com um objeto do tipo `HttpStatus`, que é um `enum` contendo todos os códigos HTTP possíveis. Como queremos utilizar o código HTTP `400 Bad Request`, utilizamos então o valor `HttpStatus.BAD_REQUEST` na parametrização dessa anotação:

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

// Restante do código omitido

@ExceptionHandler(MethodArgumentNotValidException::class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
```

```
fun handleMethodArgumentNotValid(ex: MethodArgumentNotValidException):
ErrorResponse {
    val messages = ex.bindingResult.fieldErrors.map {
        ErrorData(it.defaultMessage!!)
    }
    return ErrorResponse(messages)
}
```

Finalmente, podemos testar outra vez a API. Ao realizar novamente a requisição com o Postman, temos o seguinte resultado:

```
{
    "errors": [
        {
            "message": "??? null"
        }
    ]
}
```

Esse resultado é semelhante ao que esperávamos conseguir, mas a mensagem ainda não está de acordo com o que desejamos. Para alterá-la, basta voltarmos até a classe `TravelRequestInput` e preencher o atributo `message` da anotação `@NotNull`. Vamos preencher com a mensagem que utilizamos antes:

```
data class TravelRequestInput(
    @get:NotNull(message = "O campo passengerId não pode ser nulo")
    val passengerId: Long?,

    val origin: String?,
    val destination: String?
)
```

Ao testar mais uma vez, obtemos o resultado desejado:

```
{
    "errors": [
        {
            "message": "O campo passengerId não pode ser nulo"
        }
```

```
    ]
}
```

Um último teste que podemos fazer é para conferir o quão flexível nosso sistema é. Podemos adicionar validações nos campos `origin` e `destination` e verificar se as mensagens correspondentes são lançadas sem que precisemos realizar mais alterações no restante do código. Sabemos que esses dois campos precisam estar preenchidos para que a requisição de viagem seja criada, então vamos utilizar a anotação `@NotEmpty` sobre eles (não esquecendo de customizar o campo `message`):

```
import javax.validation.constraints.NotEmpty
import javax.validation.constraints.NotNull

data class TravelRequestInput(
    @get:NotNull(message = "O campo passengerId não pode ser nulo")
    val passengerId: Long?,

    @get:NotEmpty(message = "O campo origin não pode estar em branco")
    val origin: String?,

    @get:NotEmpty(message = "O campo destination não pode estar em
branco")
    val destination: String?
)
```

Vamos testar novamente utilizando um JSON quase inteiramente vazio:

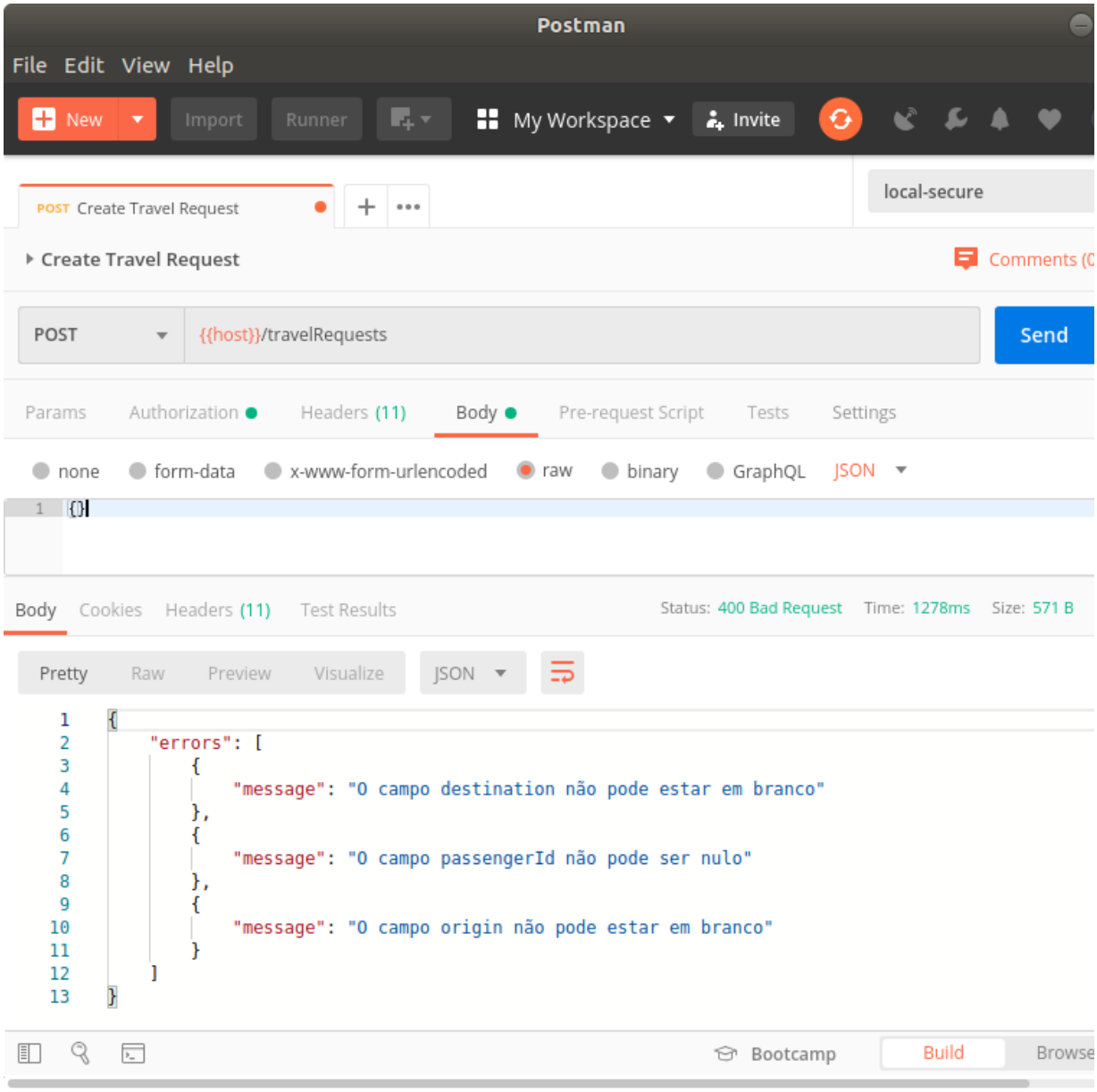


Figura 7.1: A request sendo completamente validada

7.4 Internacionalizando as mensagens de erro

Uma feature também muito interessante de se ter em uma API REST é a internacionalização. Esse efeito é produzido quando o cliente informa o idioma em que deseja que as mensagens do

sistema estejam traduzidas. Por exemplo: digamos que eu queira que o sistema C.A.R. tenha mensagens produzidas em português e inglês americano. Neste caso, o protocolo HTTP padroniza a forma como essa solicitação é feita para o servidor.

Isso é feito utilizando-se o cabeçalho `Accept-Language`, que é enviado pelo cliente. Ele obedece às regras de formação da seguinte maneira:

- Os valores fornecidos podem ser mais ou menos específicos. Por exemplo, é possível solicitar português (valor sendo `pt`) ou português do Brasil (`pt-BR`).
- Os valores fornecidos são separados por vírgula. Por exemplo, para solicitar português do Brasil ou inglês, utilizamos `pt-BR, en`.
- É possível dar um peso para a ordem de preferência. O peso padrão é sempre `1.0`; caso queira que um valor tenha menos precedência, informe um valor abaixo de 1 (porém, acima de 0) com o atributo `q`. Exemplo: `pt-BR, en-US; q=0.8, en; q=0.6`. Neste caso, a preferência será do português brasileiro, depois do inglês dos EUA e, por último, qualquer inglês disponível).

Perceba que o *parse* desse cabeçalho pode ser um tanto quanto complicado de se fazer manualmente - primeiro, é preciso interpretar as ordens de preferência e, depois, filtrar a lista de acordo com os idiomas disponíveis no sistema. Felizmente, o Spring nos disponibiliza estruturas que facilitam todos esses passos.

O LOCALE

Costumamos nos referir a estruturas de localização como *locales*, que podem ser mais ou menos específicos de acordo com a necessidade. Um *locale* pode fornecer informações sobre língua, país e até dialetos de regiões específicas deste país. Por exemplo, o *locale* do Norueguês padrão Nynorsk é `no-NO-NY` (a Noruega tem dois padrões de idioma, o Nynorsk e o Bokmål). O `no` é relativo ao idioma norueguês, o `NO` é o código da Noruega e o `NY` é o código do dialeto Nynorsk.

O primeiro passo é criar uma pasta no sistema para que possamos armazenar os arquivos que vão conter as mensagens do sistema. Vamos chamar essa pasta de `i18n` (uma contração de *internationalization*, ou internacionalização):

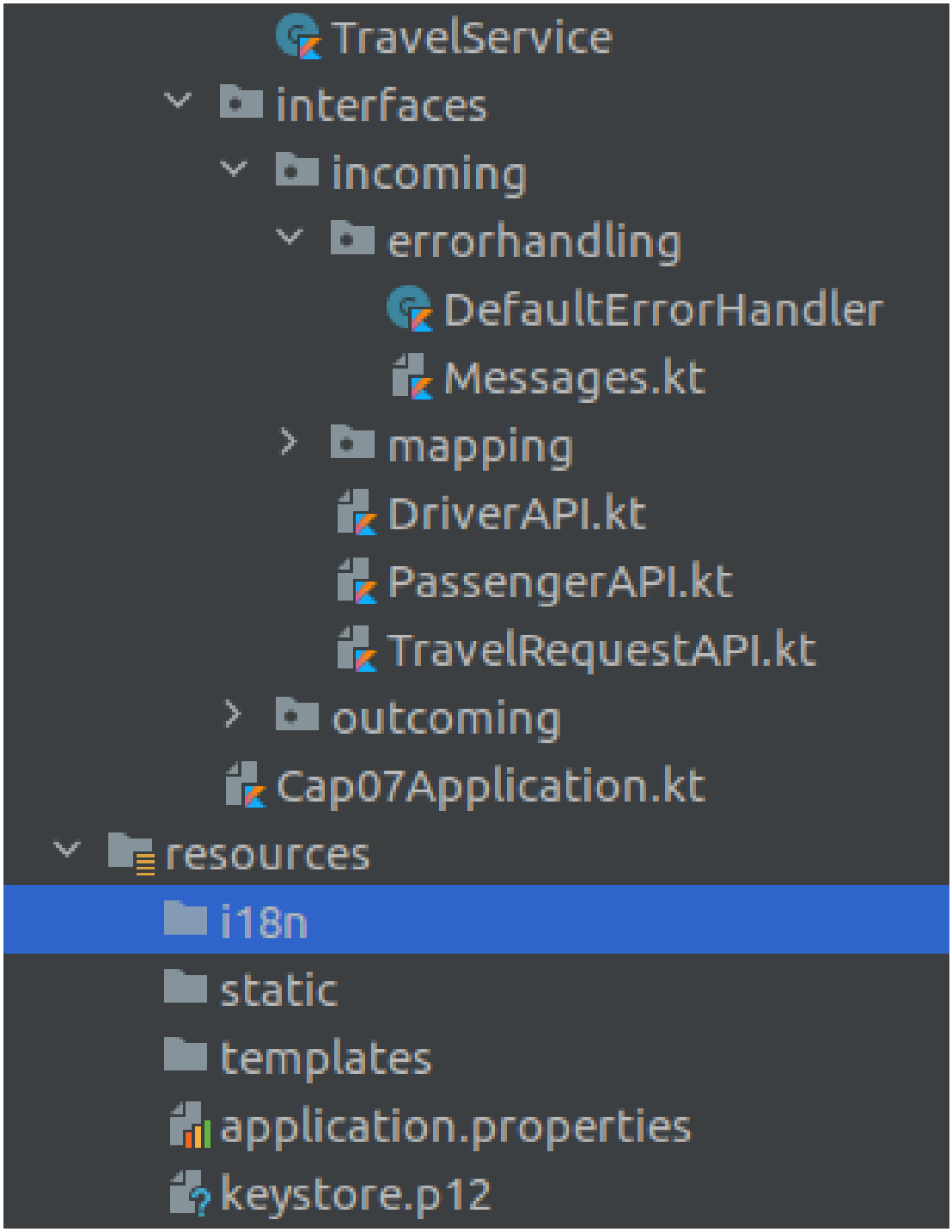


Figura 7.2: Estrutura do projeto

O próximo passo é criar dois arquivos nessa pasta que vão representar as mensagens das quais precisamos. Um deles vai ser chamado `messages_en.properties` (o prefixo `messages` e mais o *locale*

`en`). O segundo vai ser chamado `messages_pt_BR.properties` (o prefixo `messages` com o *locale* `pt_BR`):

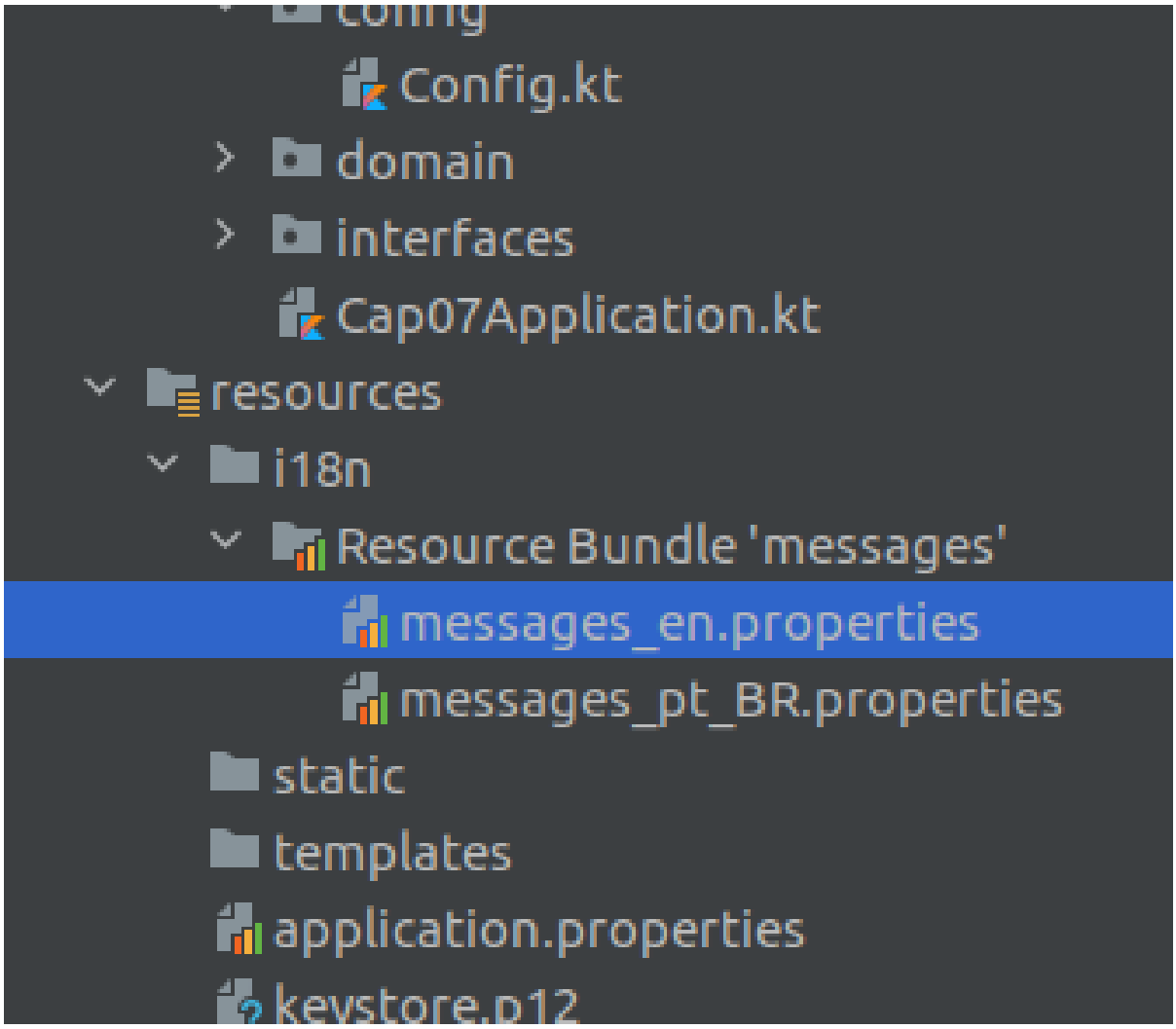


Figura 7.3: Arquivos properties

Precisamos agora utilizar a classe adequada para gerenciar esses arquivos. Dentro do Spring, utilizamos uma interface para lidar com mensagens chamada `MessageSource`. Vamos criar uma classe de configuração do sistema que forneça uma implementação dessa interface. Assim, vamos criar uma classe chamada `AppConfig` (dentro do arquivo `Config.kt`) e anotá-la com `@Configuration`. Na sequência, vamos criar um método `messageSource()` que retorne um `MessageSource` e anotá-lo com `@Bean`:

```
@Configuration
class AppConfig {

    @Bean
    fun messageSource(): MessageSource? = null
}
```

Agora, precisamos fazer com que esse método retorne uma implementação de `MessageSource`. Essa interface possui muitas implementações, mas a que mais se aproxima das nossas necessidades é a `ReloadableResourceBundleMessageSource`, pois ela lê os arquivos que estão no *classpath* e consegue escolher quais deles utilizar com base no *locale*. Basta que façamos a configuração do prefixo dos arquivos e ele cuida do restante. No nosso caso, como os arquivos estão na raiz do *classpath*, dentro do diretório `i18n` e têm o prefixo `messages`, vamos configurar a implementação da seguinte forma:

```
import
org.springframework.context.support.ReloadableResourceBundleMessageSource

// restante da implementação omitido

@Bean
fun messageSource() = ReloadableResourceBundleMessageSource().apply {
    setBasename("classpath:/i18n/messages")
}
```

Vamos agora implementar uma forma de extrair o *locale* correto do *header* `Accept-Language`. Apenas recapitulando, as "preocupações" que devemos ter com essa extração são as seguintes:

- Uma lista de *locales* pode ser enviada. Caso o atributo `q` esteja presente em algum deles, também é necessário levar em consideração o reposicionamento dentro da lista de preferências.
- Alguns *locales* podem não ser suportados por nossa aplicação.
- É possível enviar um curinga (`*`) no cabeçalho, indicando que qualquer *locale* é aceito.

Tendo isso em mente, saiba que o Spring nos disponibiliza uma classe que resolve boa parte desse processo, a chamada `AcceptHeaderLocaleResolver`. Essa classe define um método chamado `resolveLocale`, que retorna um objeto do tipo `Locale`. Esse objeto estará acessível através da classe `LocaleContextHolder`, método `getLocale`, que é estático. Por isso, é possível recuperar o *locale* correto a partir de qualquer ponto do código.

Vamos então iniciar a implementação do método `resolveLocale`. Ele recebe como parâmetro um objeto do tipo `HttpServletRequest` e retorna um `Locale`, assim:

```
package app.car.cap07.interfaces.incoming.errorhandling

import org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver
import java.util.Locale
import javax.servlet.http.HttpServletRequest

class LocaleResolver: AcceptHeaderLocaleResolver() {

    override fun resolveLocale(request: HttpServletRequest): Locale {
        return super.resolveLocale(request)
    }
}
```

Vamos anotar essa classe com `@Component` para que ela fique disponível para o Spring automaticamente:

```
import org.springframework.stereotype.Component

// Restante do código omitido

@Component
class LocaleResolver: AcceptHeaderLocaleResolver() {

    override fun resolveLocale(request: HttpServletRequest): Locale {
        return super.resolveLocale(request)
    }
}
```

Agora criaremos duas constantes na nossa classe para determinar que nosso *locale* padrão é o português do Brasil mas que também podemos fornecer mensagens em inglês. A primeira constante se chamará `DEFAULT_LOCALE` e será uma instância de `Locale` com os valores `pt` e `BR`. A outra se chamará `ACCEPTED_LOCALES` e vai conter tanto o `DEFAULT_LOCALE` quanto uma nova instância de `Locale` parametrizada apenas com `en`:

```
package app.car.cap07.interfaces.incoming.errorhandling;

import java.util.Locale

// Restante dos imports omitido

@Component
class LocaleResolver: AcceptHeaderLocaleResolver() {

    private val DEFAULT_LOCALE = Locale("pt", "BR")

    private val ACCEPTED_LOCALES = listOf(DEFAULT_LOCALE, Locale("en"))

    override fun resolveLocale(request: HttpServletRequest): Locale {
        return super.resolveLocale(request)
    }
}
```

Vamos à implementação do método. Utilizaremos o método `getHeader` do objeto `HttpServletRequest` para recuperar o valor do *header* `Accept-Language`. Este *header* pode não ser enviado pelo cliente, ser enviado em branco ou ser enviado com o curinga (`*`). Para lidar com esses aspectos, vamos implementar a análise do cabeçalho usando os métodos `isNullOrEmpty` e `trim`:

```
override fun resolveLocale(request: HttpServletRequest): Locale {
    val acceptLanguageHeader = request.getHeader("Accept-Language")
    if (acceptLanguageHeader.isNullOrEmpty() ||
acceptLanguageHeader.trim() == "*") {
        return DEFAULT_LOCALE
    }
```

```
    return null
}
```

E vamos usar a classe `Locale.LanguageRange` para fazer *parse* do valor do cabeçalho. O método `parse` vai retornar uma lista de `Locale.LanguageRange`, que é uma representação de cada `Locale` já levando em consideração a preferência por cada valor (que é determinada pelo conteúdo do atributo `q`). Depois de fazer isso, podemos submeter tanto a lista de `Locale.LanguageRange` quanto a lista de *locales* suportados para o método `lookup` da classe `Locale`. Esse método compara as duas listas para selecionar qual `Locale` melhor se encaixa nas nossas necessidades, e retorna `null` caso não existam opções na lista. Nesse caso, vamos usar o Elvis operator para retornar o Locale padrão:

```
override fun resolveLocale(request: HttpServletRequest): Locale {
    val acceptLanguageHeader = request.getHeader("Accept-Language")
    if (acceptLanguageHeader.isNullOrEmpty() ||
acceptLanguageHeader.trim() == "*") {
        return DEFAULT_LOCALE
    }
    val list = Locale.LanguageRange.parse(acceptLanguageHeader)
    return Locale.lookup(list, ACCEPTED_LOCALES) ?: DEFAULT_LOCALE
}
```

A classe finalizada fica assim:

```
package app.car.cap07.interfaces.incoming.errorhandling

import org.springframework.stereotype.Component
import org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver
import java.util.Locale
import javax.servlet.http.HttpServletRequest

@Component
class LocaleResolver: AcceptHeaderLocaleResolver() {

    private val DEFAULT_LOCALE = Locale("pt", "BR")

    private val ACCEPTED_LOCALES = listOf(DEFAULT_LOCALE, Locale("en"))
```

```
    override fun resolveLocale(request: HttpServletRequest): Locale {
        val acceptLanguageHeader = request.getHeader("Accept-Language")
        if (acceptLanguageHeader.isNullOrEmpty() ||
acceptLanguageHeader.trim() == "*") {
            return DEFAULT_LOCALE
        }
        val list = Locale.LanguageRange.parse(acceptLanguageHeader)
        return Locale.lookup(list, ACCEPTED_LOCALES) ?: DEFAULT_LOCALE
    }
}
```

Vamos ajustar a nossa classe `DefaultErrorHandler` para que os códigos de erro sejam ajustados para o `Locale`. Começaremos revendo a implementação do método `handleMethodArgumentNotValid`:

```
@ExceptionHandler(MethodArgumentNotValidException::class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
fun handleMethodArgumentNotValid(ex: MethodArgumentNotValidException):
ErrorResponse {
    val messages = ex.bindingResult.fieldErrors.map {
        ErrorData(it.defaultMessage!!)
    }
    return ErrorResponse(messages)
}
```

Nossa tarefa é modificar a linha onde o método `map` é chamado para que, em vez de instanciar um `ErrorData` neste momento, possamos fazer a chamada para um novo método que será criado. Assim sendo, vamos criar o método `getMessage`, que recebe um `FieldError` como parâmetro, e retornar um `ErrorData`:

```
@ExceptionHandler(MethodArgumentNotValidException::class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
fun handleMethodArgumentNotValid(ex: MethodArgumentNotValidException):
ErrorResponse {
    val messages = ex.bindingResult.fieldErrors.map {
        getMessage(it)
    }
    return ErrorResponse(messages)
}
```

```
fun getMessage(fieldError: FieldError) : ErrorData {
    return null
}
```

Como vimos antes, cada `FieldError` tem uma lista de `codes` que vai desde um código que é mais específico até um genérico. Vejamos os códigos que foram retornados anteriormente:

```
"codes": [
   "NotNull.travelRequestInput.passengerId",
   "NotNull.passengerId",
   "NotNull.java.lang.Long",
   "NotNull"
]
```

Dessa forma, é possível criar uma validação para quando o campo `passengerId` da classe `travelRequestInput` for nulo; quando apenas o campo `passengerId` for nulo (de qualquer classe); quando qualquer campo do tipo `java.lang.Long` for nulo e, finalmente, quando qualquer campo for nulo. Com base nesses códigos, podemos criar mensagens de validação para qualquer campo, mas vamos criar especificamente para o caso mais amplo possível, que é quando o código for `NotNull` .

Para isso, vamos abrir os arquivos `messages_pt_BR.properties` e `messages_en.properties` e criar valores para a chave `NotNull` , cada um deles em um idioma específico. Assumindo que o português do Brasil vai ser o nosso idioma padrão para as mensagens, vamos primeiro abrir o arquivo `messages_pt_BR.properties` e colocar o seguinte conteúdo:

```
NotNull=O campo não pode ser nulo
```

E vamos proceder de forma semelhante com o arquivo `messages_en.properties` :

```
NotNull=The field must not be null
```

Resta um último detalhe: se várias mensagens similares forem retornadas, não vamos conseguir saber a qual campo elas se referem. É possível parametrizar as mensagens de erro colocando índices para os parâmetros. Vamos novamente alterar o conteúdo dos arquivos da seguinte forma:

```
NotNull=O campo {0} não pode ser nulo
```

E

```
NotNull=The field {0} must not be null
```

Para implementar a busca do código, vamos injetar o `MessageSource` na classe:

```
package app.car.cap07.interfaces.incoming.errorhandling;

import org.springframework.context.MessageSource;

//Restante dos imports omitido

@RestControllerAdvice
class DefaultErrorHandler(
    val messageSource: MessageSource
) {

    // Restante do código omitido
}
```

A interface `MessageSource` oferece três implementações do método `getMessage`:

- A primeira recebe o código da mensagem, um *array* de parâmetros para a mensagem, uma String com a mensagem padrão e o `Locale`. Caso a mensagem não seja encontrada, retorna a mensagem padrão.
- A segunda recebe o código da mensagem, um *array* de parâmetros e o `Locale`. Caso a mensagem não seja encontrada, lança uma `NoSuchMessageException`.

- A terceira recebe uma implementação da interface `MessageSourceResolvable` e o `Locale`.

No nosso caso, podemos aproveitar que o `FieldError` implementa a interface `MessageSourceResolvable`. Dessa forma, tudo o que temos a fazer é utilizar o método `getMessage` passando como parâmetro o `FieldError` e o `Locale`. Este último pode ser obtido a partir da classe `LocaleContextHolder`, que provê um método chamado `getLocale` (por baixo dos panos, ele vai retornar o resultado do método que codificamos na classe `LocaleResolver`). Toda a classe finalizada fica assim:

```
package app.car.cap07.interfaces.incoming.errorhandling

import org.springframework.context.MessageSource
import org.springframework.context.i18n.LocaleContextHolder
import org.springframework.http.HttpStatus
import org.springframework.validation.FieldError
import org.springframework.web.bind.MethodArgumentNotValidException
import org.springframework.web.bind.annotation.ExceptionHandler
import org.springframework.web.bind.annotation.ResponseStatus
import org.springframework.web.bind.annotation.RestControllerAdvice

@RestControllerAdvice
class DefaultErrorHandler(
    val messageSource: MessageSource
) {

    @ExceptionHandler(MethodArgumentNotValidException::class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    fun handleMethodArgumentNotValid(ex: MethodArgumentNotValidException):
ErrorResponse {
        val messages = ex.bindingResult.fieldErrors.map {
            getMessage(it)
        }
        return ErrorResponse(messages)
    }

    fun getMessage(fieldError: FieldError) : ErrorData {
        return ErrorData(messageSource.getMessage(fieldError,
```

```
LocaleContextHolder.getLocale()))
    }
}
```

Vamos testar novamente com o Postman, passando primeiro o *locale* `en` no cabeçalho `Accept-Language`:

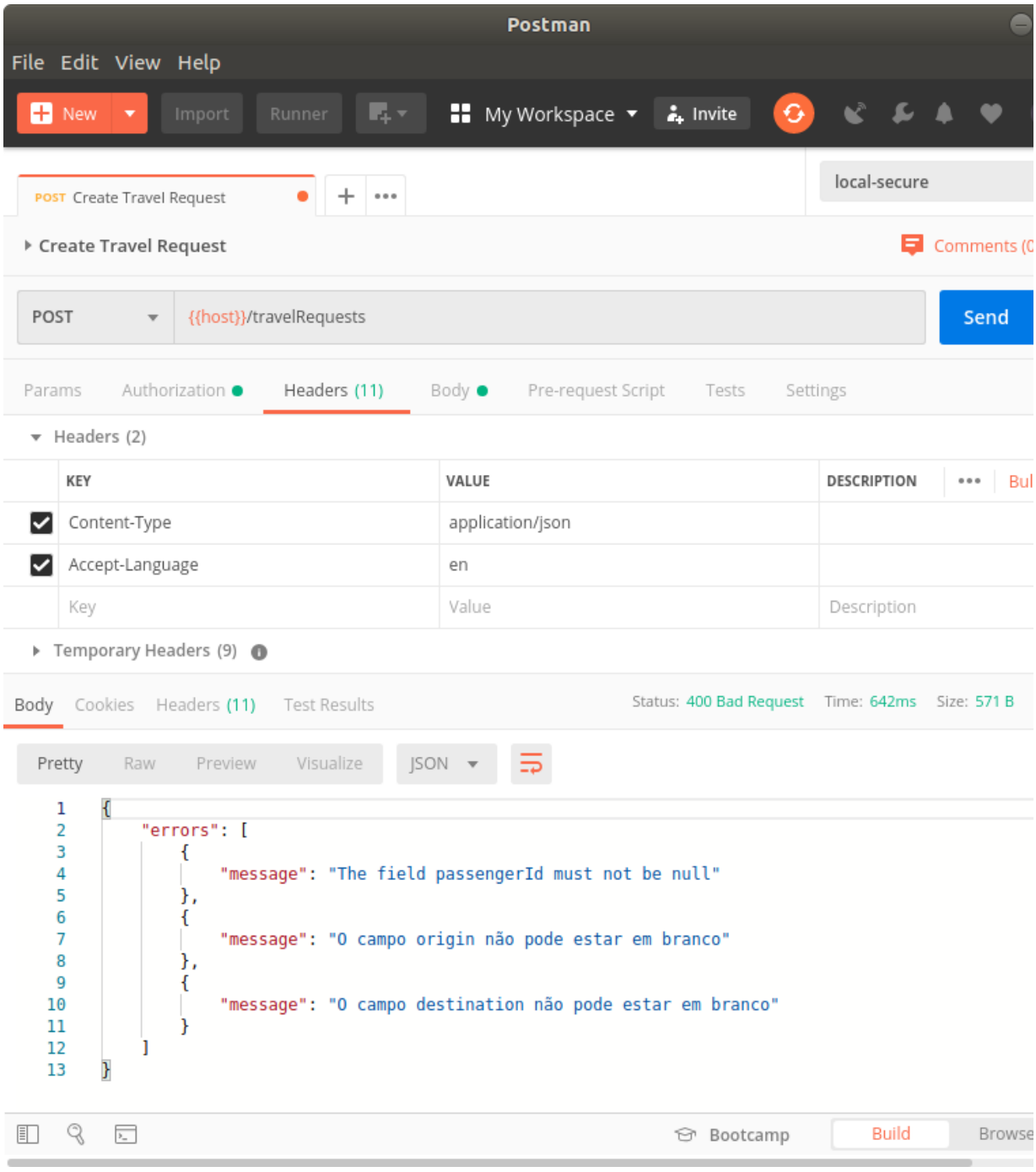


Figura 7.4: Teste com Locale en

Observe que a mensagem do `passengerId` veio em inglês, enquanto as outras duas vieram em português. Isso acontece, porque não configuramos a mensagem da validação da anotação `@NotEmpty` e o

sistema usou as mensagens padrão, ou seja, as que estavam diretamente nas anotações.

Vamos abrir novamente os arquivos de propriedades e configurar as mensagens, começando pelo `messages_pt_BR.properties` :

```
NotNull=O campo {0} não pode ser nulo
NotEmpty=O campo {0} não pode estar em branco
```

Agora o arquivo `messages_en.properties` :

```
NotNull=The field {0} must not be null
NotEmpty=The field {0} must not be empty
```

Vamos testar novamente:

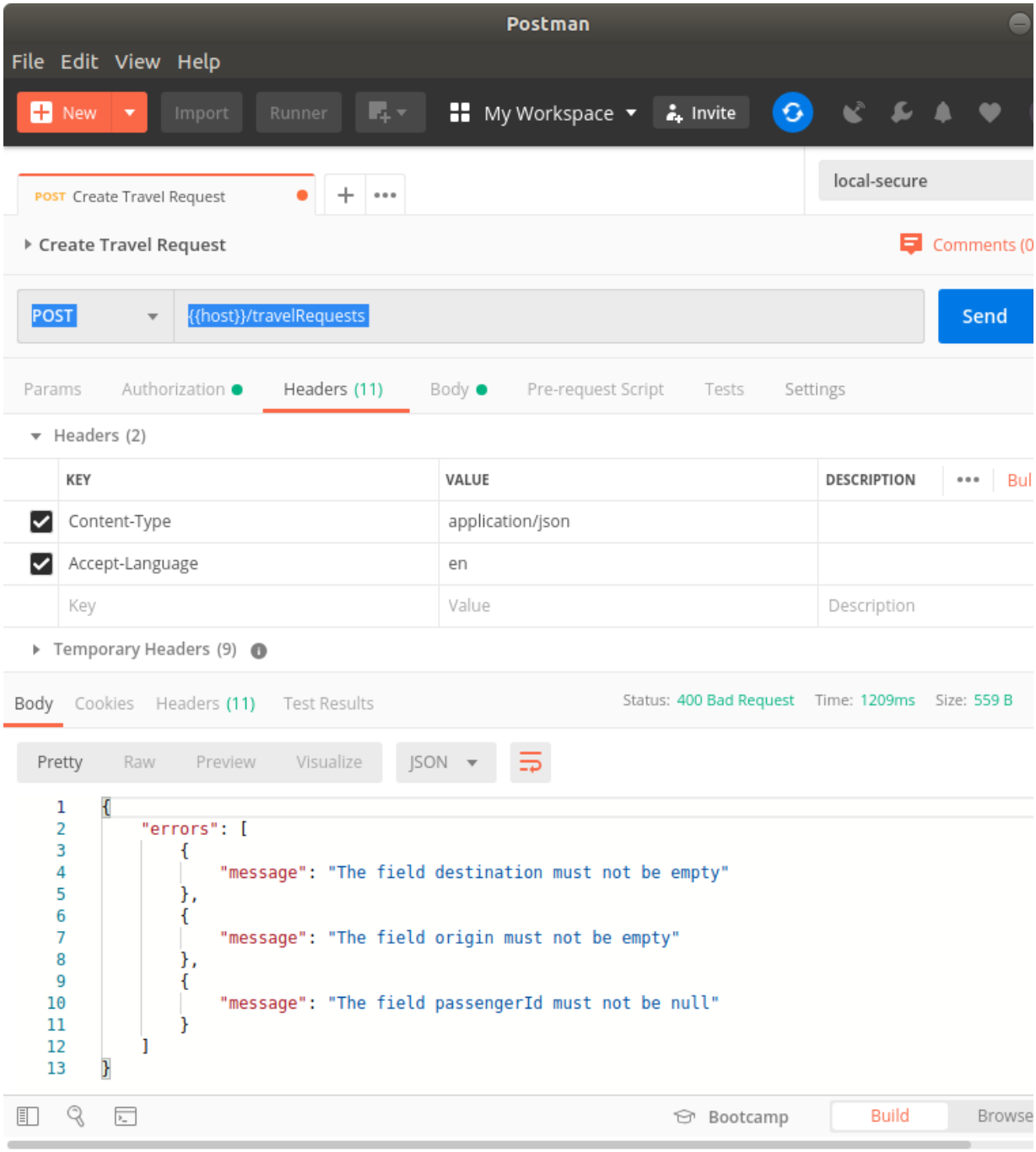


Figura 7.5: Novo teste com Locale en

Agora, vamos ajustar o cabeçalho para *locale* `pt-BR` :

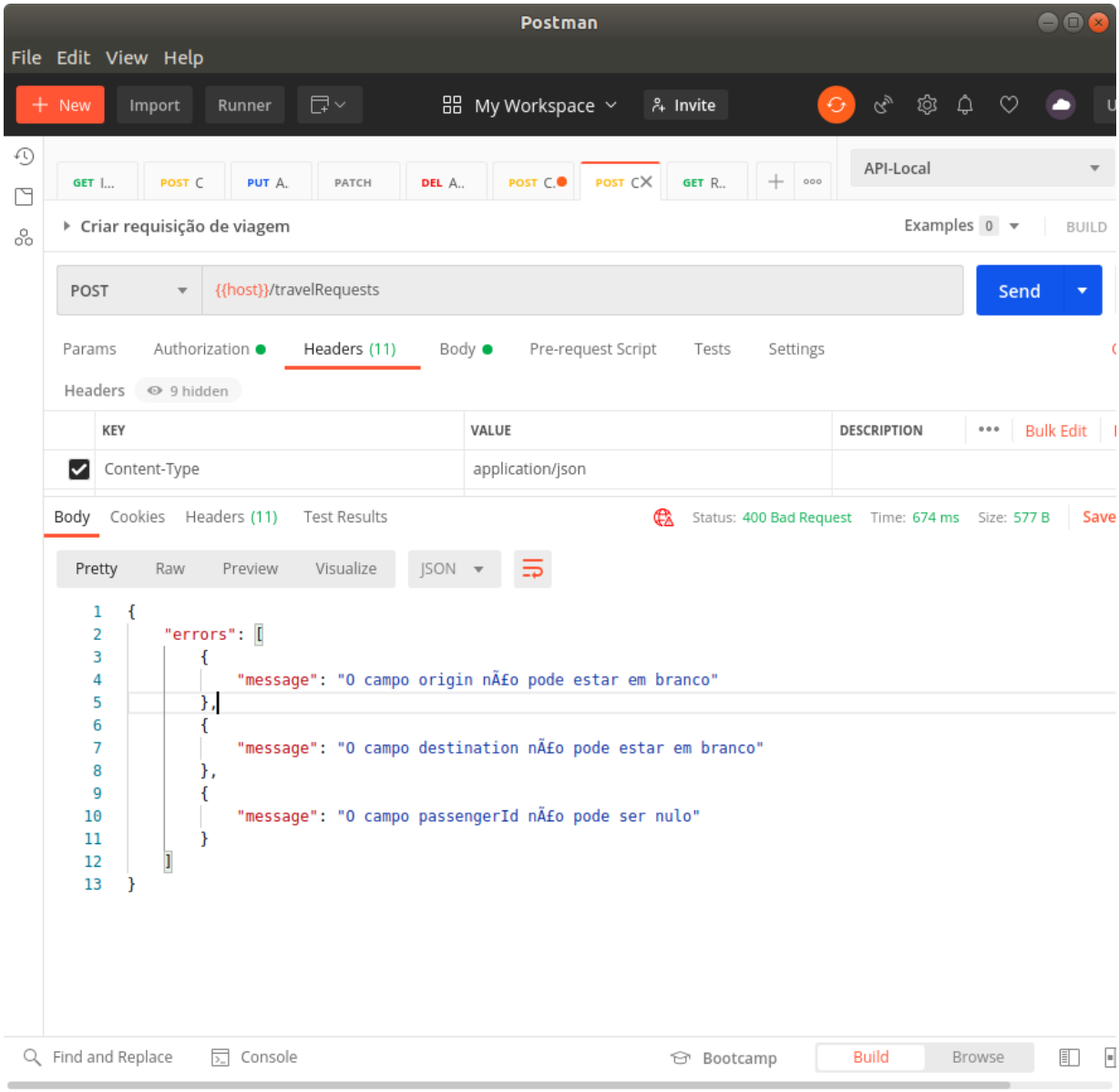


Figura 7.6: Teste com Locale pt-BR

Observe que os caracteres acentuados não foram corretamente codificados. Resolver isso é simples, basta substituir os caracteres acentuados pelas suas representações em Unicode. No nosso caso, o arquivo `messages_pt_BR.properties` fica assim:

```
NotNull=O campo {0} n\u00E3o pode ser nulo
NotEmpty=O campo {0} n\u00E3o pode estar em branco
```

Vamos testar novamente:

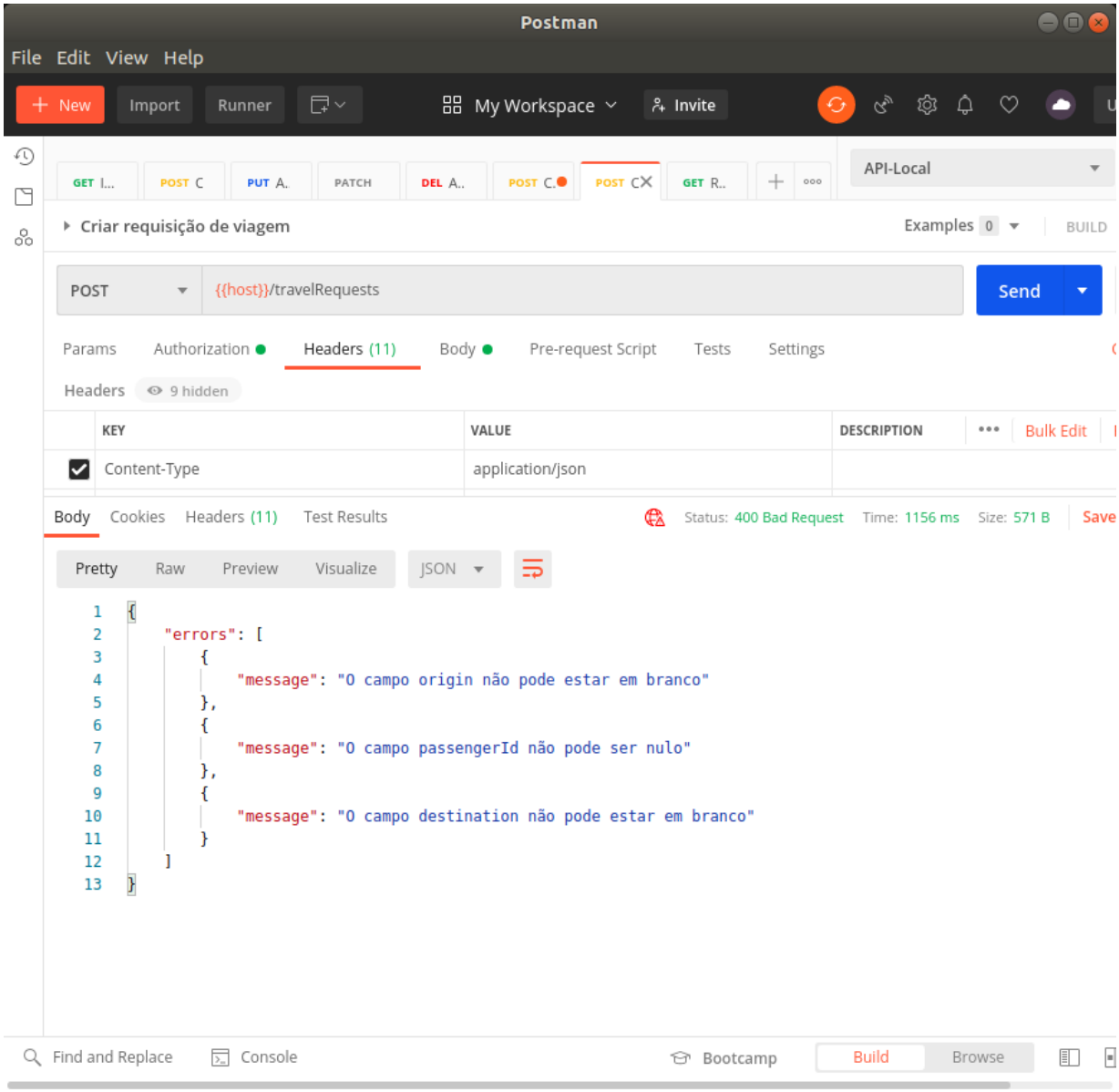


Figura 7.7: Novo teste com Locale pt-BR ajustado

Agora ficou bom. O último teste que precisamos fazer é com um *locale* não utilizado no sistema. Vamos colocar `zh` (Chinês) para verificar o que acontece:

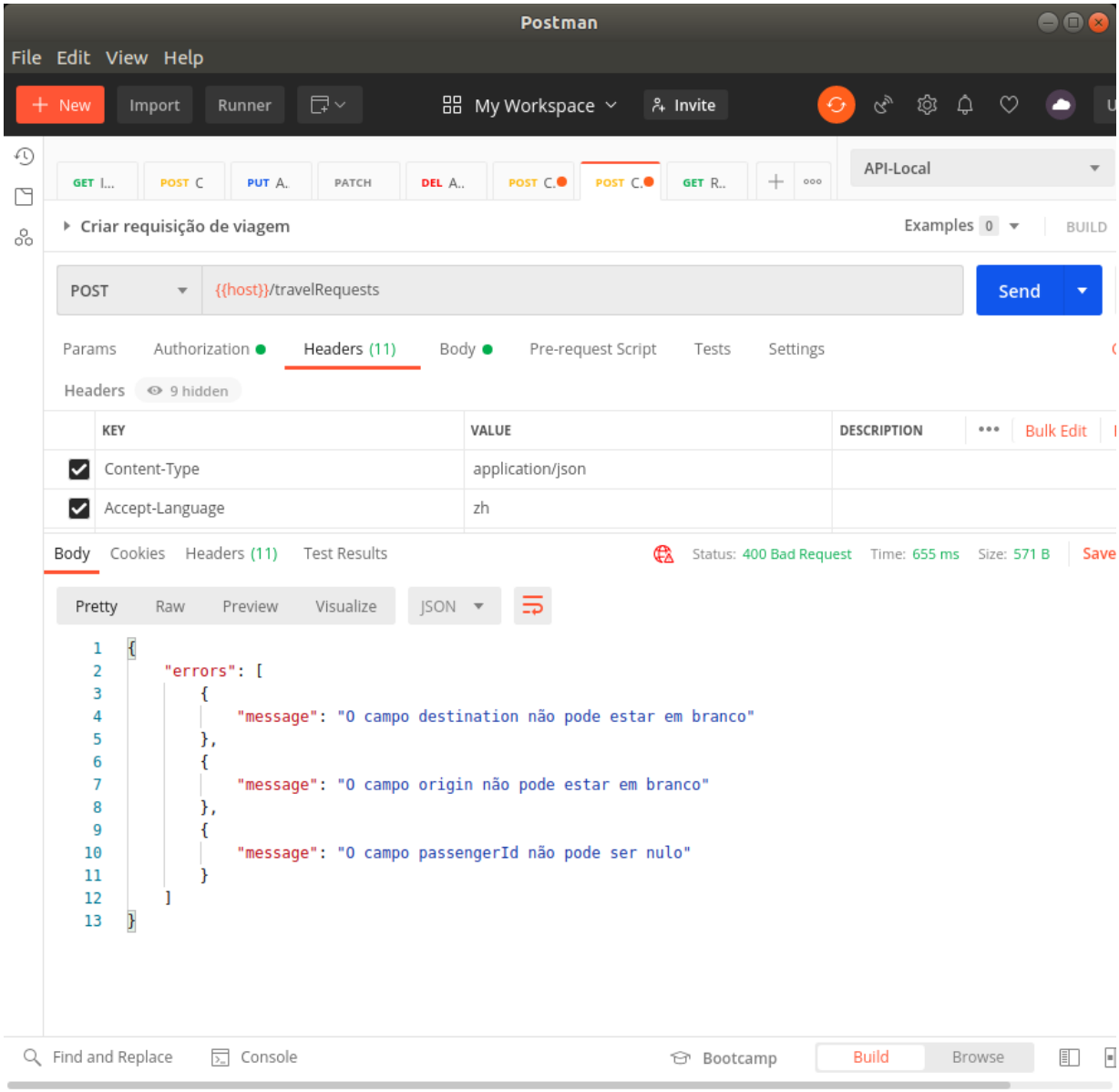


Figura 7.8: Teste com Locale zh

Por que isso aconteceu? Quando codificamos o nosso `LocaleResolver`, na hora de fazer o `lookup`, o *locale* chinês não estava na nossa lista de *locales* suportados. Assim, o método `lookup` retornou nulo e isso acionou o Elvis operator, que fez com que o *locale* de português retornasse.

7.5 Como criar uma API retrocompatível (ou como versionar uma API)

Nossa API está crescendo. É provável que o negócio em si a acompanhe no crescimento, e isso vai gerando novas demandas para a API. Com novas demandas, a API precisa ser evoluída/melhorada. Isso faz com que uma API muitas vezes não possa mais ser usada.

Vamos ver um exemplo na própria API que construímos, `/travelRequests`. Uma *request* típica seria assim:

```
{
   "passengerId": 1,
   "destination": "Avenida Faria Lima, 1300",
   "origin": "Avenida Paulista, 100"
}
```

Se o time de negócios decidisse que, além do endereço, precisamos do CEP em um campo separado, poderíamos desenhar uma nova *request* para ser mais ou menos da seguinte forma:

```
{
   "passengerId": 1,
   "destination": {
      "name":"Avenida Faria Lima, 1300",
      "zipCode":"05426-100"
   },
   "origin": "Avenida Paulista, 100"
}
```

Mas existe um problema fundamental aí: os clientes atuais da API esperam que o campo `destination` seja uma String, não um objeto mais complexo. Isso faria com que a API deixasse de funcionar para esses clientes já existentes, e faria com que ela não fosse **retrocompatível**. A retrocompatibilidade é muito importante em ambientes onde você não conhece os usuários da sua API ou você não tem controle sobre eles. Por exemplo, se você tiver um

aplicativo *mobile* que consulta a sua API, pode ser que os usuários do aplicativo passem um bom tempo sem atualizar a versão dele mas, mesmo assim, você deseja que essas versões mais antigas continuem funcionando sem forçar a atualização.

Quando uma API é criada para ser retrocompatível, algumas regras devem ser seguidas:

- Não se pode modificar campos simples para se tornarem complexos.
- Não se pode introduzir novos campos obrigatórios.
- Não se pode criar regras restringindo valores aceitos em um campo, por exemplo, tomar um campo que aceite até cinco caracteres e fazer com que passe a aceitar até três.
- Não se pode introduzir qualquer espécie de parâmetro obrigatório.
- Não se pode alterar as URLs.
- Não se pode alterar nomes de campos, tanto na *request* quanto na *response*.

Como via de regra, é mais fácil dizer o que pode ser feito em uma API que **não** provoca erros na retrocompatibilidade:

- Criação de novos campos não obrigatórios.
- Flexibilização de regras de validação, por exemplo, um campo que aceita até cinco caracteres passa a aceitar dez.
- Criação de novas URLs na API.

Como você pode observar, é mais difícil deixar uma API retrocompatível do que o contrário. Essa técnica de manutenção de uma API para que não haja quebra dos clientes é semelhante ao incremento de uma *minor version* em um software qualquer. Se fôssemos mesmo atribuir uma versão à nossa API, poderíamos dizer que é esperado que os clientes feitos para utilizar a versão `1.0` da API continuem funcionando corretamente quando a API vai para a versão `1.1`, sem qualquer alteração. O mesmo já não pode ser dito se a API for para uma suposta versão `2.0`: a *major version*

é incrementada quando há quebra da compatibilidade - logo, não sendo retrocompatível.

Não é desejável haver quebra da compatibilidade; portanto implementamos uma estratégia de **versionamento da API** para que ao menos possamos manter versões da API executando em paralelo. Mantê-las em paralelo nos permite criar estratégias para executar um planejamento de *phase out* nos clientes, ou seja, avisá-los de que eles possuem um certo prazo para atualizar a API para a versão mais recente enquanto a antiga não deixa de funcionar.

No aspecto técnico, para versionar as APIs, criamos uma forma do cliente informar ao servidor qual versão da API gostaria de usar. As principais formas de fazer isso são as seguintes:

- Modificar a URL dos serviços para incluir a versão. Exemplo: `/v1/travelRequests` .
- Criar um *query param* para fornecer a versão. Exemplo: `/travelRequests?version=1` .
- Criar um *header* para fornecer a versão.

Cada uma dessas formas tem vantagens e desvantagens, que vamos investigar mais a fundo a seguir.

Versionamento através de mudança de hostnames

Exemplo: https://v1.api.car.com.br/travelRequests

Esta estrutura é a mais flexível de todas, mas é a que tem mais "potencial para desastre". Com essa estrutura é possível recriar por completo uma API - pois, se o *host* muda, é como se fosse uma nova integração. Porém, ao forçar o cliente a criar essa nova integração, todo um *overhead* gerado por esse processo é assumido. Parte desse *overhead* significa configurar possíveis servidores *proxy*, regras de roteamento Apache/nginx etc. Contudo, do lado de quem provê a API, essas mesmas dificuldades podem ser transformadas em facilidades, pois permitem que as APIs estejam localizadas em máquinas ou *datacenters* diferentes, e até

mesmo que sejam escritas em linguagens distintas. Efetivamente, é como oferecer APIs totalmente novas.

Pró:

- Permite reestruturar por completo a estrutura.

Contra:

- Pode prejudicar demais os clientes quando a mudança é realizada, talvez até os forçando a manter estruturas diferentes.

Versionamento pela URL

Exemplo: https://api.car.com.br/v1/travelRequests

Esta é uma das técnicas mais adotadas quando se fala a respeito de versionamento de APIs, por ser uma das mais diretas. A API que é versionada desta forma pode decidir se força o cliente a informar a versão ou não. Por exemplo, a mesma API pode estar disponível em **https://api.car.com.br/v1/travelRequests** e **https://api.car.com.br/travelRequests**, sendo que, nesta última, o servidor adota a última versão como padrão. Tal feito não precisa necessariamente ser codificado diretamente na aplicação; basta que o fornecedor da API crie uma regra no balanceador de carga, como o já mencionado nginx. Isso pode ou não ser fácil/conveniente, mas é uma opção caso não seja desejável criar o roteamento diretamente na aplicação.

Um aspecto que torna delicada a adoção desse modelo é o uso de HATEOAS, pois assim o cliente não tem a flexibilidade para escolher quais versões utilizar de quais links, isto é, ele pode ser forçado por um link HATEOAS a utilizar uma versão diferente da desejada. O mesmo vale para a estratégia de, caso a versão não esteja presente, utilizar a última: um link que tenha uma natureza mais permanente pode criar o mesmo conflito no cliente, onde este referencia um link contando com uma determinada versão e esta passa por *upgrade* internamente.

Em todos os casos citados, fica claro também que a coexistência de recursos com diferentes versões pode ser complexa, uma vez que um serviço pode passar por um *upgrade* e outro, não. Dessa forma, em um cenário com vários serviços convivendo em um mesmo ecossistema fica clara a necessidade de se ter alguma forma de gerenciamento das versões dos serviços.

Por último, deixo claro que, conforme explanado ao longo deste livro, o versionamento por URLs não é RESTful. Isso significa que os serviços que adotam esse modelo de versionamento não estão 100% de acordo com a especificação REST original (FIELDING, 2000). Assim sendo, tal uso de versionamento pode acabar levando as APIs a destoarem do modelo como um todo e paulatinamente perderem a característica da usabilidade.

> **O QUE SIGNIFICA A PALAVRA RESTFUL?**
>
> O sufixo **ful** depois de REST é uma forma da língua inglesa de adaptar ou transformar um substantivo em um adjetivo. Para ficar mais claro, temos como exemplos a palavra **color** (cor) e **colorful** (colorido). Da mesma forma, entende-se REST como um substantivo e RESTful como um adjetivo. Na maior parte dos contextos, pode ser usada de maneira intercambiável.

Prós:

- Torna a versão utilizada óbvia.
- Fácil de criar regras de roteamento.

Contras:

- Quebra HATEOAS.
- Quebra links permanentes.
- Não é RESTful.
- Coexistência entre recursos de versões diferentes.

Versionamento por query params

Exemplo: https://api.car.com.br/travelRequests?v=1

Uma maneira substancialmente mais simples de versionar APIs seria através de *query params*. No entanto, essa facilidade não perduraria por muito tempo, pois, além de apresentar alguns dos problemas demonstrados pelo versionamento via URL, acrescenta-se um problema com o roteamento, já que alguns balanceadores de carga podem não o reconhecer como um método de roteamento.

Um problema ainda mais grave com essa abordagem é que os *query params* são historicamente utilizados apenas no método GET. É um pouco mais comum hoje em dia outros métodos suportarem, mas nota-se que essa não é uma boa prática e deve ser evitada. Observa-se que algumas bibliotecas não suportam isso, como a Fuel (de desenvolvimento Android): https://github.com/kittinunf/fuel/issues/231.

Pró:

- Método simples.

Contras:

- Difícil de criar regras de roteamento.
- Nem todos os métodos HTTP e bibliotecas suportam.
- Não é RESTful.

Versionamento por um cabeçalho customizado

Exemplo:

Accept-Version: v1;q=0.8,v2;q=0.6

https://api.car.com.br/travelRequests

Ao passo que o versionamento por URL não é RESTful, este mecanismo o é. A passagem de cabeçalhos no protocolo HTTP demonstra como funciona o tráfego de metadados entre o cliente e

o fornecedor, e o uso de um cabeçalho para informar a versão é uma excelente demonstração disso.

Para efeitos de comparação, lembre-se de que temos outros cabeçalhos de negociação de conteúdo disponíveis na própria especificação HTTP: `Accept`, `Accept-Language`, `Accept-Encoding` e outros.

Assim sendo, um cabeçalho do tipo `Accept-Version` (ou similar) estaria muito próximo daquilo que já está embutido no protocolo HTTP, ainda mais se ele tivesse o caráter **negociável** (conforme ocorre com o `Accept-Language`, por exemplo).

Isso teria várias vantagens. A primeira delas é que, ao utilizar HATEOAS ou links permanentes, o usuário teria plena possibilidade de determinar novas versões utilizando os mesmos links, criando total flexibilidade.

No entanto, o fato de ser customizado faz com que o usuário precise ser informado explicitamente sobre o que inserir como informação de versão. Ele pode ainda não inserir qualquer informação e seguir a abordagem de utilizar a última versão (tendo os mesmos problemas que foram explicitados na seção de versionamento pela URL). Além disso, o próprio fornecimento do cabeçalho não é óbvio, já que os cabeçalhos costumam seguir uma natureza mais opcional no HTTP.

Os dois problemas poderiam ser resolvidos ao mesmo tempo se o cabeçalho fosse tornado obrigatório. Isso faria com que a API perdesse um pouco da sua usabilidade, mas ganharia mais resiliência nesse aspecto do versionamento.

Prós

- É RESTful.
- Pode oferecer um modelo mais flexível de negociação de versões.

Contras

- Não é padronizado/óbvio.
- Deixa o cliente livre para não especificar qual a versão.
- Pode quebrar HATEOAS.
- Pode apresentar inconsistências entre versões diferentes de recursos.

Versionamento pelo cabeçalho Accept

Exemplo:

Accept: application/vnd.travelrequest+json;version=1.0

https://api.car.com.br/travelRequests

Em termos de elegância, este é o melhor formato de todos, pois ele utiliza o próprio mecanismo de negociação de conteúdo REST para negociar também a versão. Embora perca em termos da negociação de versão que pode ser embutida em um cabeçalho customizado, esse formato ganha na padronização.

Entretanto, por requerer customização sobre os próprios *media types* (uma vez que as informações de versão são anexadas a estes, conforme visto no exemplo), essa vantagem pode ser rapidamente transformada em desvantagem ao passo que pode ser prejudicial para algumas bibliotecas de clientes.

Prós

- É o formato mais RESTful.
- Torna mais óbvia a questão do versionamento.

Contras

- Pode não ser completamente compatível com as bibliotecas de clientes.
- Pode interferir no funcionamento correto da negociação de conteúdo.

- Não oferece tanta flexibilidade a respeito de qual versão utilizar.

Finalmente, qual modelo escolher?

Todos os formatos vistos possuem vantagens e desvantagens - no fim das contas, trata-se de analisar o seu ecossistema em termos de ferramentas utilizadas, clientes etc. e verificar qual é o melhor formato. No meu ponto de vista, a abordagem com o cabeçalho customizado parece a opção com menos pontos contra - ou ao menos, a opção em que é mais fácil contornar os contras.

Conclusão

Este capítulo tem a missão de ser um divisor de águas dentro do livro. Até aqui, abordamos as questões mais básicas e óbvias de desenvolvimento de APIs com todo o básico necessário para realizar o desenvolvimento. Falamos também sobre um conteúdo mais avançado, o que realmente faz com que uma API seja fácil de usar e esteja pronta para os desafios que aparecerão uma vez que o grande público passe a utilizá-la. Mas esse foi apenas o primeiro passo; precisamos falar mais sobre outras questões, como documentação, monetização e como abordar questões como o *throttling*. Vamos em frente?

Capítulo 8
Documentando a API

"Embora a memória e o raciocínio sejam duas faculdades essencialmente diferentes, uma só se desenvolve completamente com a outra." - Jean-Jacques Rousseau.

Uma das partes mais importantes de se criar e manter uma API é a sua documentação. Afinal, se uma API é a interface para que um programa interaja com outro da melhor forma possível, a documentação é a ferramenta que garante que *pessoas* aprendam a usar a API da melhor forma possível.

Uma boa documentação precisa ser suficientemente inteligível para atingir seu público-alvo em cheio. Algumas são dirigidas a pessoas com vieses mais técnicos, como desenvolvedores, arquitetos, analistas etc. Já outras têm como alvo pessoas com vieses mais relacionados ao negócio, como *product owners*, gerentes e até algumas do chamado C-level. Isso pode provocar diferenciação nos meios utilizados para documentar a API; as documentações produzidas para pessoal técnico precisam de riqueza de detalhes técnicos e facilidade de interação. Já as documentações produzidas para as pessoas de negócio detalham com riqueza o propósito das APIs que estão sendo tratadas por elas e como o usuário final pode ser impactado.

Um grande desafio para manter uma boa documentação de APIs é a rapidez com que ela pode ficar defasada. Isso geralmente acontece quando a documentação é gerada de forma dissociada do código, isto é, quem desenvolve a API realiza a manutenção e, quando publicada, atualiza a API. Esta segunda etapa, em meio às preocupações do dia a dia, pode facilmente ser deixada de lado, gradualmente tornando a documentação inútil por não mais refletir a realidade.

Para contornar o problema, alguns frameworks foram criados ao longo do tempo e surgiu o conceito de **documentação viva**. A documentação chamada dessa forma é gerada junto ao código (muitas vezes, gerada *pelo* código), de forma que o código e sua documentação não podem ser dissociados. Isso faz com que a atualização do código e da documentação seja uma coisa só, e a documentação permaneça sempre atualizada, já que a tarefa de atualizá-la não é separada da tarefa de atualizar o código que a acompanha.

8.1 Criando uma documentação viva com Swagger/OpenAPI

Em Java, uma forma muito prática para se atingir uma documentação viva é através do uso de anotações. Um framework que adotou essa técnica com maestria foi o *Swagger* (disponível em https://swagger.io/), que desde 2010 está disponível para realizar essa tarefa. Em 2015, a especificação Swagger foi doada para a Linux Foundation e renomeada para OpenAPI, tendo em vista o objetivo de padronizar a maneira como APIs REST são documentadas. Hoje, é um dos frameworks mais populares para criação de documentação viva.

Para o Swagger começar a funcionar, vamos incluir a dependência dele no `pom.xml` do nosso projeto:

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-ui</artifactId>
   <version>1.2.32</version>
</dependency>
```

Além disso, como havíamos alterado nosso projeto para proteger todas as URLs, precisamos criar regras especiais para dispensar as URLs específicas do Swagger de autenticação. As URLs são:

- `/swagger-ui.html` .
- `/swagger-ui/` e quaisquer outras URLs derivadas.
- `/v3/api-docs/` e derivadas.

Para realizar a liberação, vamos abrir novamente o método `void configure(HttpSecurity http)` na classe `SecurityConfig` . Ele deve estar com o seguinte conteúdo:

```
override fun configure(http: HttpSecurity) {
    http.csrf().disable()

    http.sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)

    http
        .authorizeRequests()
        .anyRequest()
        .authenticated()
        .and()
        .httpBasic()
}
```

Vamos alterar a última declaração (a que contém `authorizeRequests`) para permitir que as URLs do Swagger sejam liberadas. Primeiro, precisamos incluir uma chamada para `antMatchers` , que recebe as diversas URLs com uma notação de padrões. Com esses padrões, as URLs que vão incluir derivadas podem utilizar `**` para refletir essa necessidade. Por exemplo, `/swagger-ui/` vai ficar `/swagger-ui/**` . Então, a chamada fica assim:

```
http
   .authorizeRequests()
      .antMatchers("/swagger-ui.html", "/swagger-ui/**", "/v3/api-
docs/**")
   .anyRequest()
      .authenticated()
      .and()
      .httpBasic()
```

Por último, incluímos apenas uma chamada para `permitAll()` logo abaixo de `antMatchers` para declarar que essas URLs não precisam de autenticação. Então teremos:

```
http
   .authorizeRequests()
      .antMatchers("/swagger-ui.html", "/swagger-ui/**", "/v3/api-
docs/**")
      .permitAll()
   .anyRequest()
      .authenticated()
      .and()
      .httpBasic()
```

Isso é o básico. Inicialize a aplicação e navegue para `https://localhost:8080/swagger-ui.html` .

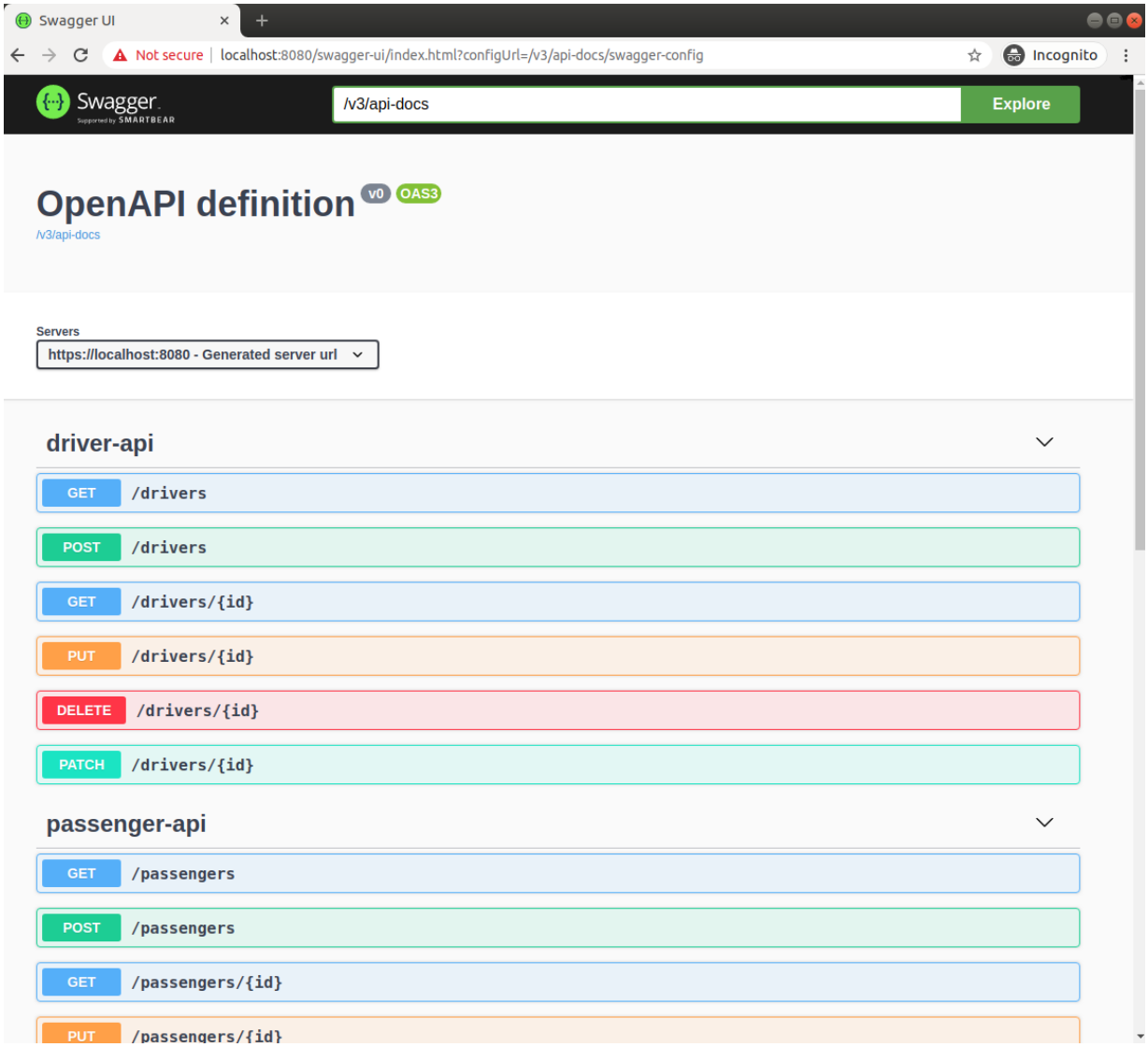


Figura 8.1: Tela inicial do Swagger

Observe que todas as APIs que criamos anteriormente já foram detectadas e estão implantadas. Por exemplo, se eu clicar na posição do `GET /drivers/{id}`, verei a seguinte tela:

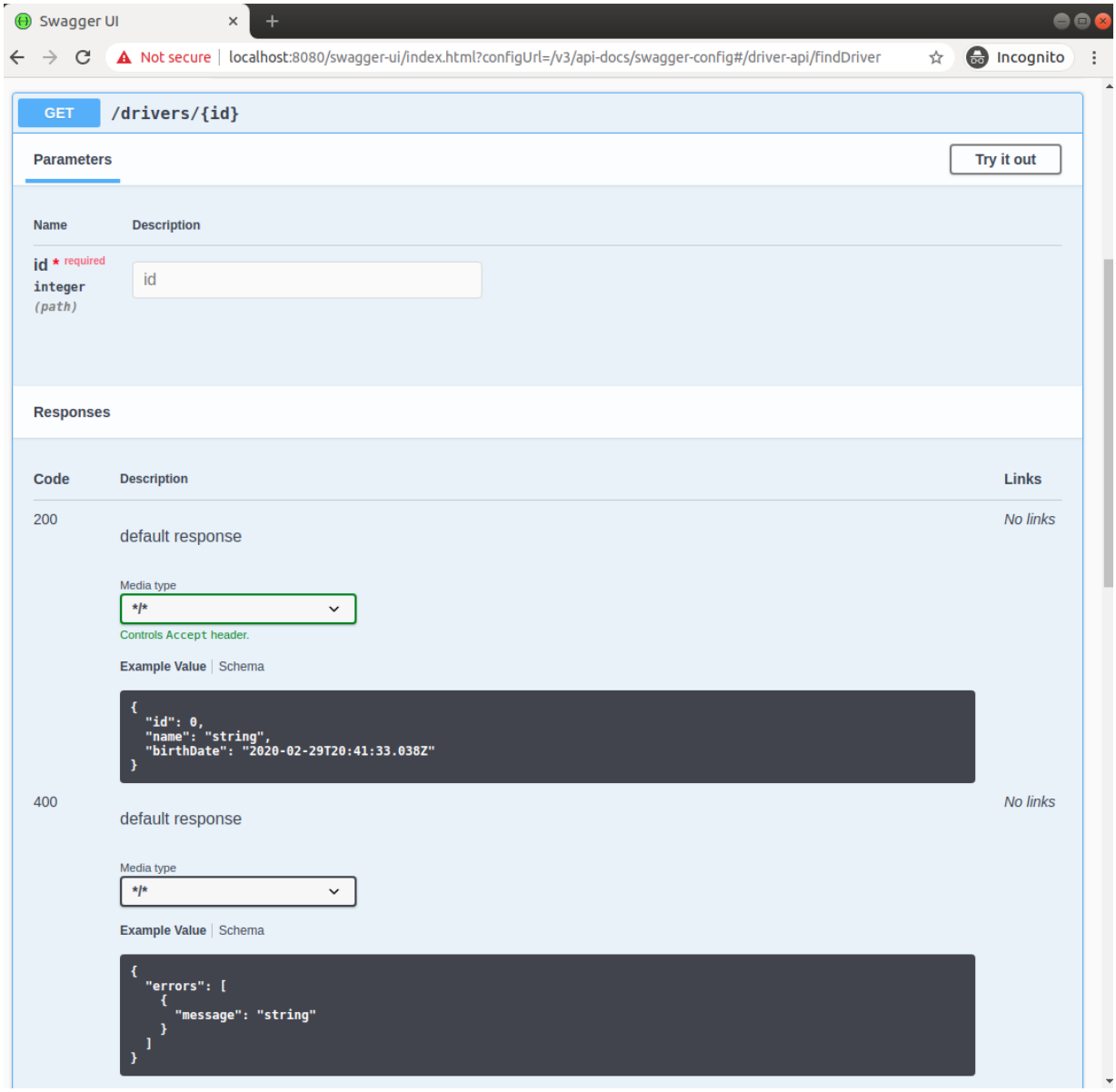


Figura 8.2: Tela do Swagger para GET em /drivers/{id}

O que falta aqui é enriquecer a documentação com descrições acerca do que representam os dados. Como já mencionado anteriormente, isso é feito através de anotações. Vamos abrir a classe `DriverAPI` e incluir a anotação `@Tag`, fornecendo os parâmetros `name` (que declara o nome da API) e `description` (que vamos usar para fornecer uma descrição detalhada do que se trata a API):

```
import io.swagger.v3.oas.annotations.tags.Tag

//Outros imports omitidos e anotações omitidas

@Tag(name = "Driver API", description = "Manipula dados de motoristas.")
class DriverAPI(
    val driverRepository: DriverRepository
) {

//Código omitido

}
```

Ao inicializar o sistema novamente e acessar a tela da OpenAPI, vemos a mudança:

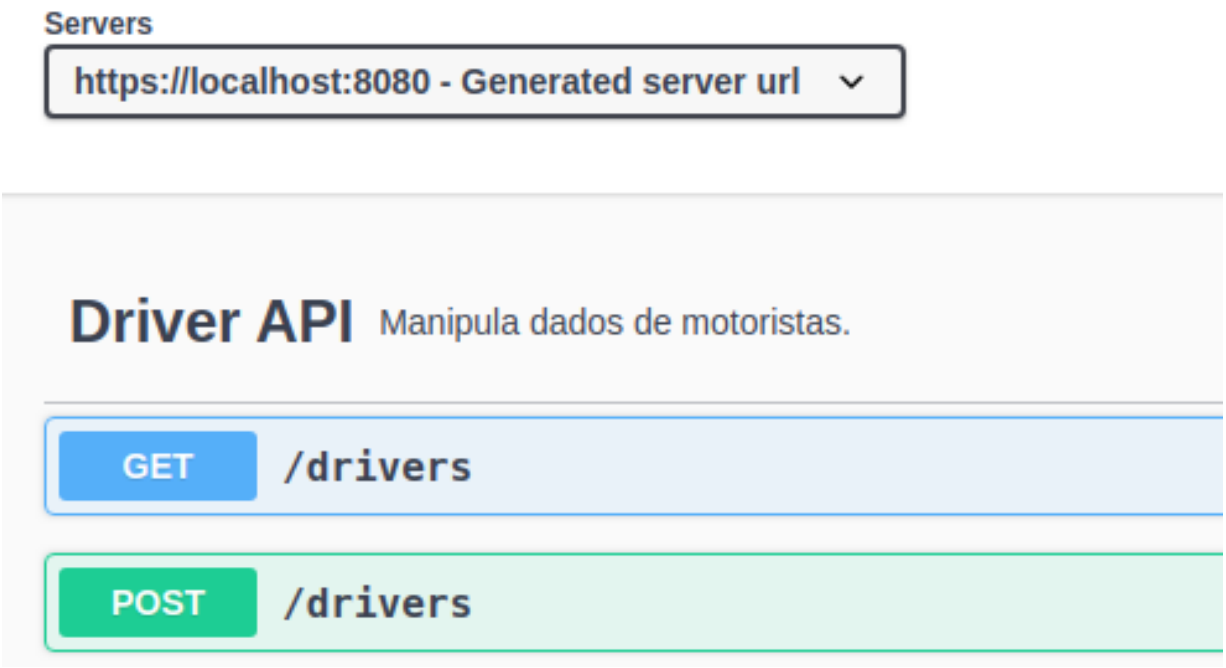


Figura 8.3: Documentação da API de motoristas mais detalhada

Para detalhar melhor cada operação da API, utilizamos a anotação `@Operation`, que tem um atributo chamado `description`. Vamos ver como isso funciona ao anotar o método `findDriver`:

```
import io.swagger.v3.oas.annotations.Operation

//Restante do código omitido


@GetMapping("/drivers/{id}")
@Operation(description = "Localiza um motorista específico")
fun findDriver(@PathVariable("id") id: Long) =
    driverRepository.findById(id)
        .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }
```

Vejamos como ficou:

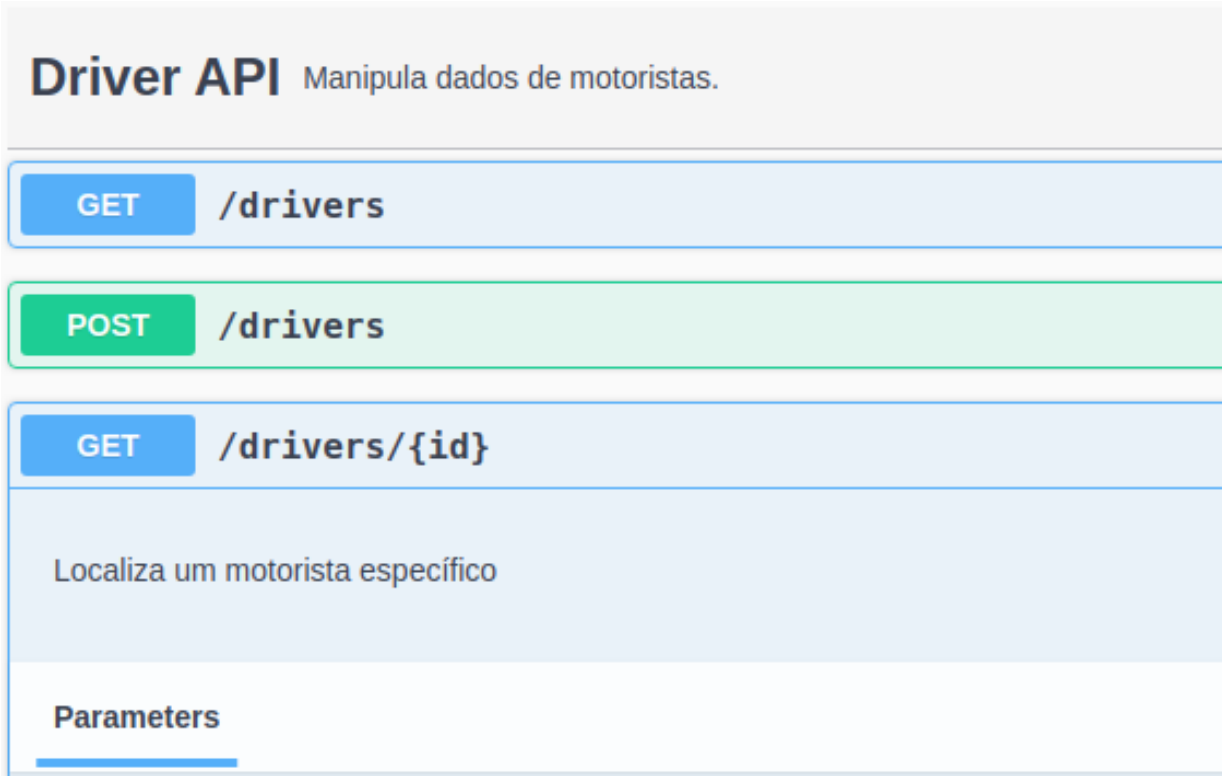


Figura 8.4: /drivers/{id} com documentação

Vamos incrementar a documentação dessa operação com as descrições dos retornos, ou seja, quais são os códigos de status utilizados nos retornos e o que é devolvido nesses casos. Vamos fazer isso utilizando o atributo `responses` da anotação `@Operation`. Como ele recebe um *array* de `@ApiResponse`, vamos utilizar os atributos desta anotação para realizar o descritivo:

```
import io.swagger.v3.oas.annotations.responses.ApiResponse


@GetMapping("/drivers/{id}")
@Operation(description = "Localiza um motorista específico", responses = [
    ApiResponse(responseCode = "200", description = "Caso o motorista
tenha sido encontrado na base"),
    ApiResponse(responseCode = "404", description = "Caso o motorista não
tenha sido encontrado")
])
fun findDriver(@PathVariable("id") id: Long) =
    driverRepository.findById(id)
        .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }
```

Isso se transforma no seguinte:

Code	Description
200	Caso o motorista tenha sido encontrado na base Media type application/json Controls `Accept` header. **Example Value** | Schema `{ "id": 0, "name": "string", "birthDate": "2020-02-29T21:46:04.901Z" }`
400	default response Media type */* **Example Value** | Schema `{ "errors": [ { "message": "string" } ] }`
404	Caso o motorista não tenha sido encontrado Media type application/json **Example Value** | Schema `{ "id": 0, "name": "string", "birthDate": "2020-02-29T21:46:04.905Z" }`

Figura 8.5: /drivers/{id} com documentação de erros, mas ainda com algo errado

Observe que o mapeamento está errado, pois ele detectou que uma resposta 404 seria dada com o retorno usual. Precisamos incrementar o mapeamento para que a resposta do status 404 esteja com o objeto de erro.

Infelizmente, fazer isso é um pouco mais prolixo. Requer o uso das anotações `@Content` e `@Schema` na seguinte forma:

```
import io.swagger.v3.oas.annotations.media.Content
import io.swagger.v3.oas.annotations.media.Schema

@GetMapping("/drivers/{id}")
@Operation(description = "Localiza um motorista específico", responses = [
    ApiResponse(responseCode = "200", description = "Caso o motorista
tenha sido encontrado na base"),
    ApiResponse(responseCode = "404", description = "Caso o motorista não
tenha sido encontrado",
        content = [Content(schema = Schema(implementation =
ErrorResponse::class))]
        )
])
fun findDriver(@PathVariable("id") id: Long) =
    driverRepository.findById(id)
        .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }
```

O que agora gera:

Code	Description

200	Caso o motorista tenha sido encontrado na base
	
	Media type
	application/json
	Controls Accept header.
	
	Example Value | Schema
	
	{
	"id": 0,
	"name": "string",
	"birthDate": "2020-02-29T21:53:10.714Z"
	}

400	default response
	
	Media type
	/
	
	Example Value | Schema
	
	{
	"errors": [
	{
	"message": "string"
	}
	]
	}

404	Caso o motorista não tenha sido encontrado
	
	Media type
	application/json
	
	Example Value | Schema
	
	{
	"errors": [
	{
	"message": "string"
	}
	]
	}

Figura 8.6: /drivers/{id} mapeado corretamente

Vamos incluir um mapeamento semelhante na classe `DefaultErrorHandler`, pois o Swagger não conseguiu detectar o tipo de conteúdo dos erros `400`. Então vamos incluir o mapeamento de forma similar ao do método `findDriver` e deixar o método `handleMethodArgumentNotValid` assim:

```
@ApiResponses(value = [
        ApiResponse(responseCode = "400", content = [Content(
                mediaType = "application/json",
                schema = Schema(implementation = ErrorResponse::class))
        ])
])
fun handleMethodArgumentNotValid(ex: MethodArgumentNotValidException):
ErrorResponse {
// ...
```

Voltando à classe `DriverAPI`, podemos também mapear o parâmetro `id` do método utilizando a anotação `@Parameter`:

```
fun findDriver(@Parameter(description = "ID do motorista a ser
localizado")
               @PathVariable("id") id: Long) =
    driverRepository.findById(id)
        .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }
```

Isso gera a documentação neste formato:

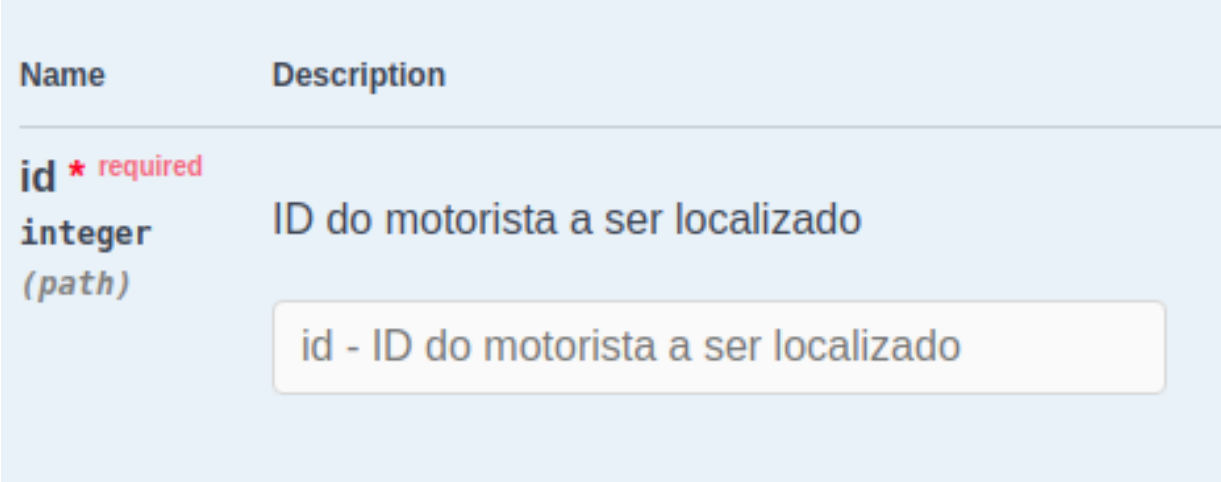


Figura 8.7: Documentação de parâmetros

Em casos de objetos mais complexos, devemos realizar o mapeamento das anotações diretamente nos objetos utilizando a anotação `@Schema` vista anteriormente. Por exemplo, a `data class Driver` fica assim:

```
@Entity
@Schema(description = "Representa um motorista dentro da plataforma")
data class Driver(
        @Id
        @GeneratedValue
        var id: Long? = null,

        @get:Schema(description = "Nome do motorista")
        val name: String,

        @get:Schema(description = "Data de nascimento do motorista")
        val birthDate: LocalDate
)
```

Para visualizar o efeito dessa alteração, vamos descer totalmente na página do Swagger, até a seção de `Schemas` :

Figura 8.8: Schema da classe Driver

Observe que o mapeamento é sensível a algumas anotações de validação, incluindo `@Size`. Isso fará com que a validação do objeto (conforme visto no capítulo 7) também seja utilizada para efeitos de documentação. Vamos alterar a `data class Driver` para ver isso na prática:

```
@Entity
@Schema(description = "Representa um motorista dentro da plataforma")
data class Driver(
        @Id
        @GeneratedValue
        var id: Long? = null,

        @get:Schema(description = "Nome do motorista")
```

```
        @get:Size(min = 5, max = 255)
        val name: String,

        @get:Schema(description = "Data de nascimento do motorista")
        val birthDate: LocalDate
)
```

O resultado fica:

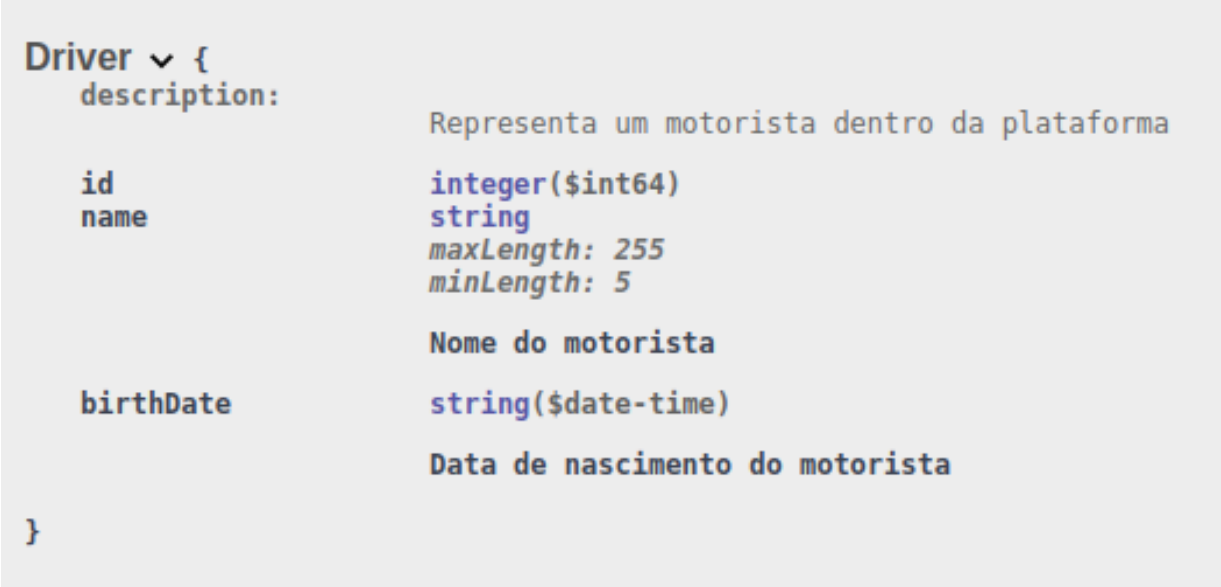


Figura 8.9: Schema da classe Driver com restrições de tamanho

O último ponto é acrescentar algumas informações sobre a documentação em si, como o nome da API, uma breve descrição, versão e informações de contato. Isso é feito através de um objeto `OpenAPI`, que nós inserimos no contexto do Spring através da estratégia de criação de configurações:

```
package app.car.cap08.config;

import io.swagger.v3.oas.models.OpenAPI
import io.swagger.v3.oas.models.info.Contact
import io.swagger.v3.oas.models.info.Info

@Configuration
class OpenAPIConfig {
```

```
    @Bean
    fun openAPIDocumentation(): OpenAPI {
        return OpenAPI()
            .info(
                Info()
                    .title("C.A.R. API")
                    .description("API do sistema C.A.R., de facilitação de
mobilidade urbana")
                    .version("v1.0")
                    .contact(
                        Contact()
                            .name("Alexandre Saudate")
                            .email("alesaudate@gmail.com")
                    )
            )
    }
}
```

O que reflete na documentação da seguinte forma:

Figura 8.10: Cabeçalho da documentação com as novas informações

A última ação a ser tomada é simplificar a classe `DriverAPI`. As novas anotações tomaram muito espaço e prejudicam a leitura da implementação - para isso, uma boa prática é criar uma interface e colocar as anotações de documentação nela. Dessa forma, renomeamos a classe para `DriverAPIImpl` e criamos uma nova interface chamada `DriverAPI`, com o seguinte corpo:

```
@Tag(name = "Driver API", description = "Manipula dados de motoristas.")
interface DriverAPI {

    @Operation(description = "Lista todos os motoristas disponíveis")
    fun listDrivers() : List<Driver>

    @Operation(description = "Localiza um motorista específico", responses
= [
        ApiResponse(responseCode = "200", description = "Caso o motorista
tenha sido encontrado na base"),
        ApiResponse(responseCode = "404", description = "Caso o motorista
não tenha sido encontrado",
            content = [Content(schema = Schema(implementation =
ErrorResponse::class))]
        )
    ])
    fun findDriver(@Parameter(description = "ID do motorista a ser
localizado") id: Long) : Driver

    fun createDriver(driver: Driver): Driver

    fun fullUpdateDriver(id:Long, driver:Driver) : Driver

    fun incrementalUpdateDriver(id:Long, driver: PatchDriver) : Driver

    fun deleteDriver(@PathVariable("id") id: Long)

}
```

Agora, podemos simplesmente retirar todas as anotações `OpenAPI` da classe `DriverAPIImpl` e fazer com que esta implemente a interface `DriverAPI`, tomando o cuidado de declarar que os métodos implementam os descritos na interface, a partir da palavra `override`.

A classe `DriverAPIImpl` fica assim:

```
@Service
@RestController
@RequestMapping(produces = [MediaType.APPLICATION_JSON_VALUE])
class DriverAPIImpl(
    val driverRepository: DriverRepository
) : DriverAPI{

    @GetMapping("/drivers")
    override fun listDrivers() = driverRepository.findAll()

    @GetMapping("/drivers/{id}")
    override fun findDriver(@PathVariable("id") id: Long) =
        driverRepository.findById(id)
            .orElseThrow { ResponseStatusException(HttpStatus.NOT_FOUND) }

    @PostMapping("/drivers")
    override fun createDriver(@RequestBody driver: Driver) =
driverRepository.save(driver)

    @PutMapping("/drivers/{id}")
    override fun fullUpdateDriver(@PathVariable("id") id:Long,
@RequestBody driver:Driver) : Driver {
        val foundDriver = findDriver(id)
        val copyDriver = foundDriver.copy(
            birthDate = driver.birthDate,
            name = driver.name
        )
        return driverRepository.save(copyDriver)
    }

    @PatchMapping("/drivers/{id}")
    override fun incrementalUpdateDriver(@PathVariable("id") id:Long,
@RequestBody driver: PatchDriver) : Driver {
        val foundDriver = findDriver(id)
        val copyDriver = foundDriver.copy(
            birthDate = driver.birthDate ?: foundDriver.birthDate,
            name = driver.name ?: foundDriver.name
        )
        return driverRepository.save(copyDriver)
```

```
    }

    @DeleteMapping("/drivers/{id}")
    override fun deleteDriver(@PathVariable("id") id: Long) =
        driverRepository.delete(findDriver(id))
}
```

8.2 Utilizando o documenter do Postman

Uma forma também muito utilizada pelos criadores de APIs como um todo para criar uma documentação acessível tanto para o pessoal de negócios quanto para o pessoal técnico é utilizar o próprio Postman. Essa maneira de gerar documentação é um pouco menos efetiva do que a OpenAPI em termos de manter-se viva, já que depende do Postman (ou seja, uma ferramenta externa ao código) para ser criada e atualizada. Entretanto, por ser uma ferramenta utilizada para se interagir de forma técnica com as APIs, essa desvantagem é parcialmente compensada.

Sua grande vantagem é manter acessível a documentação independente do código. O Postman possui uma forma de publicar a documentação e deixá-la disponível on-line, de forma que qualquer um que queira utilizar a API pode fazer isso com a documentação aberta. Ele ainda ajuda quem desenvolve mostrando como consumir cada API com várias linguagens disponíveis, compensando bastante a desvantagem de ser apartado do código.

Vamos começar atualizando a documentação. Atualizei a coleção do Postman com todos os recursos que foram criados até agora e os separei por pastas:

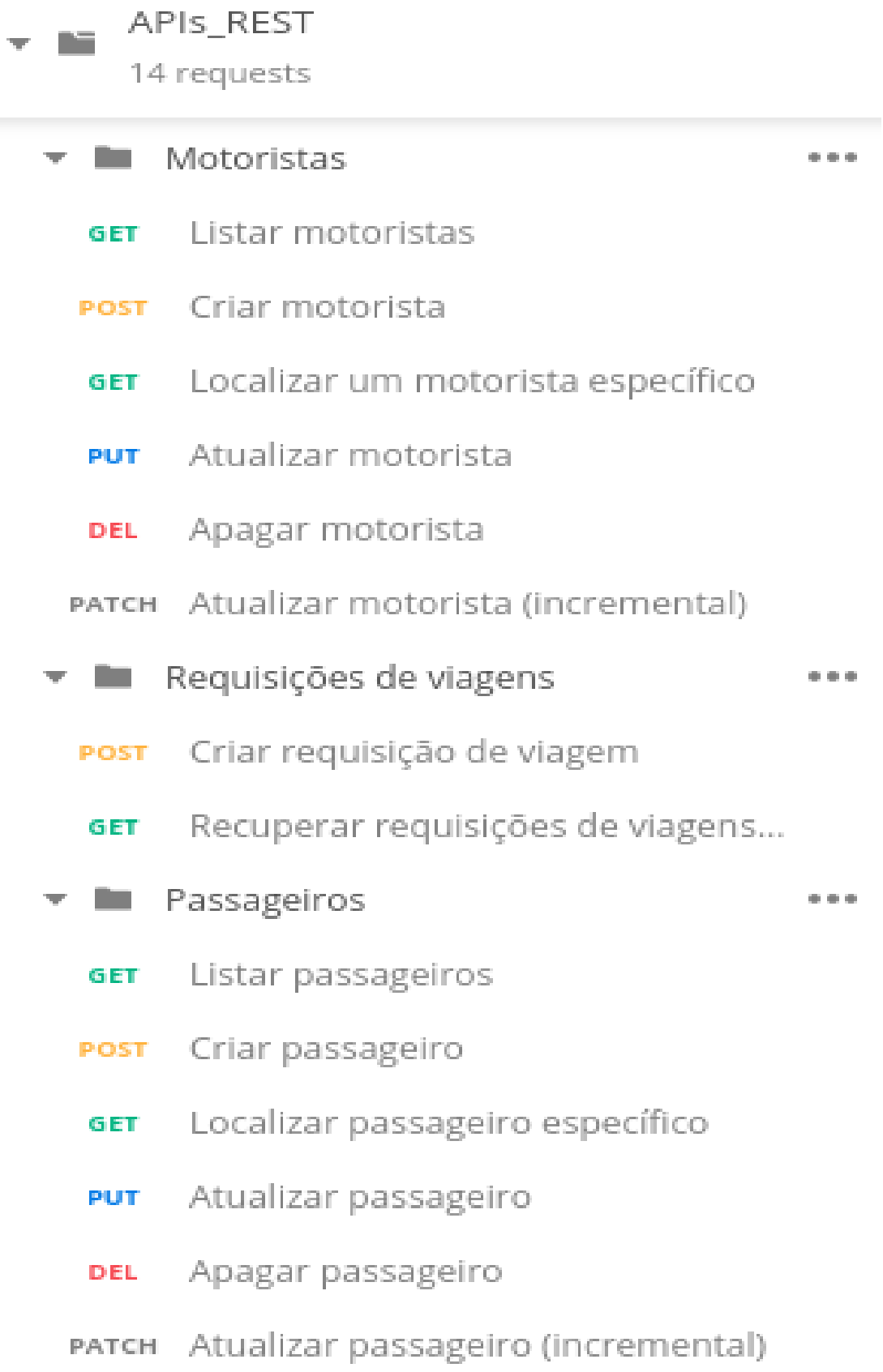

APIs_REST
14 requests
Motoristas
GET Listar motoristas
POST Criar motorista
GET Localizar um motorista específico
PUT Atualizar motorista
DEL Apagar motorista
PATCH Atualizar motorista (incremental)
Requisições de viagens
POST Criar requisição de viagem
GET Recuperar requisições de viagens...
Passageiros
GET Listar passageiros
POST Criar passageiro
GET Localizar passageiro específico
PUT Atualizar passageiro
DEL Apagar passageiro
PATCH Atualizar passageiro (incremental)

Figura 8.11: Collection do Postman completa

Ao clicar sobre o ícone de reticências da API, vejo uma série de opções:

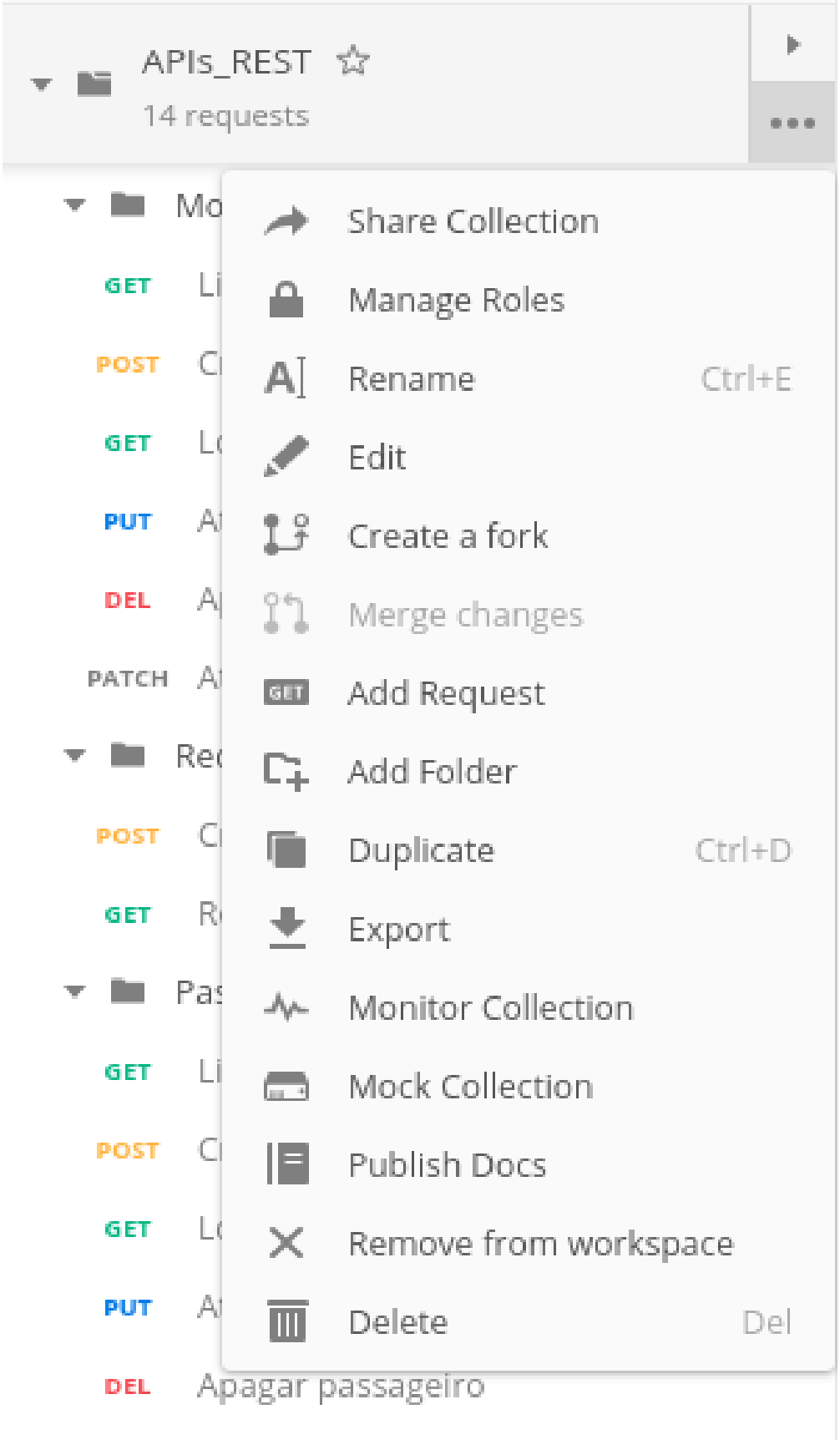

APIs_REST
14 requests
Share Collection
Manage Roles
Rename Ctrl+E
Edit
Create a fork
Merge changes
Add Request
Add Folder
Duplicate Ctrl+D
Export
Monitor Collection
Mock Collection
Publish Docs
Remove from workspace
Delete Del
GET
POST
PUT
DEL
PATCH
Apagar passageiro

Figura 8.12: Menu dropdown da collection do Postman

Para publicar a *collection*, clique no botão `Publish Docs`. Isso vai fazer com que o Postman envie essa *collection* para o site e abra uma tela no navegador para configuração de como essa documentação será exposta:

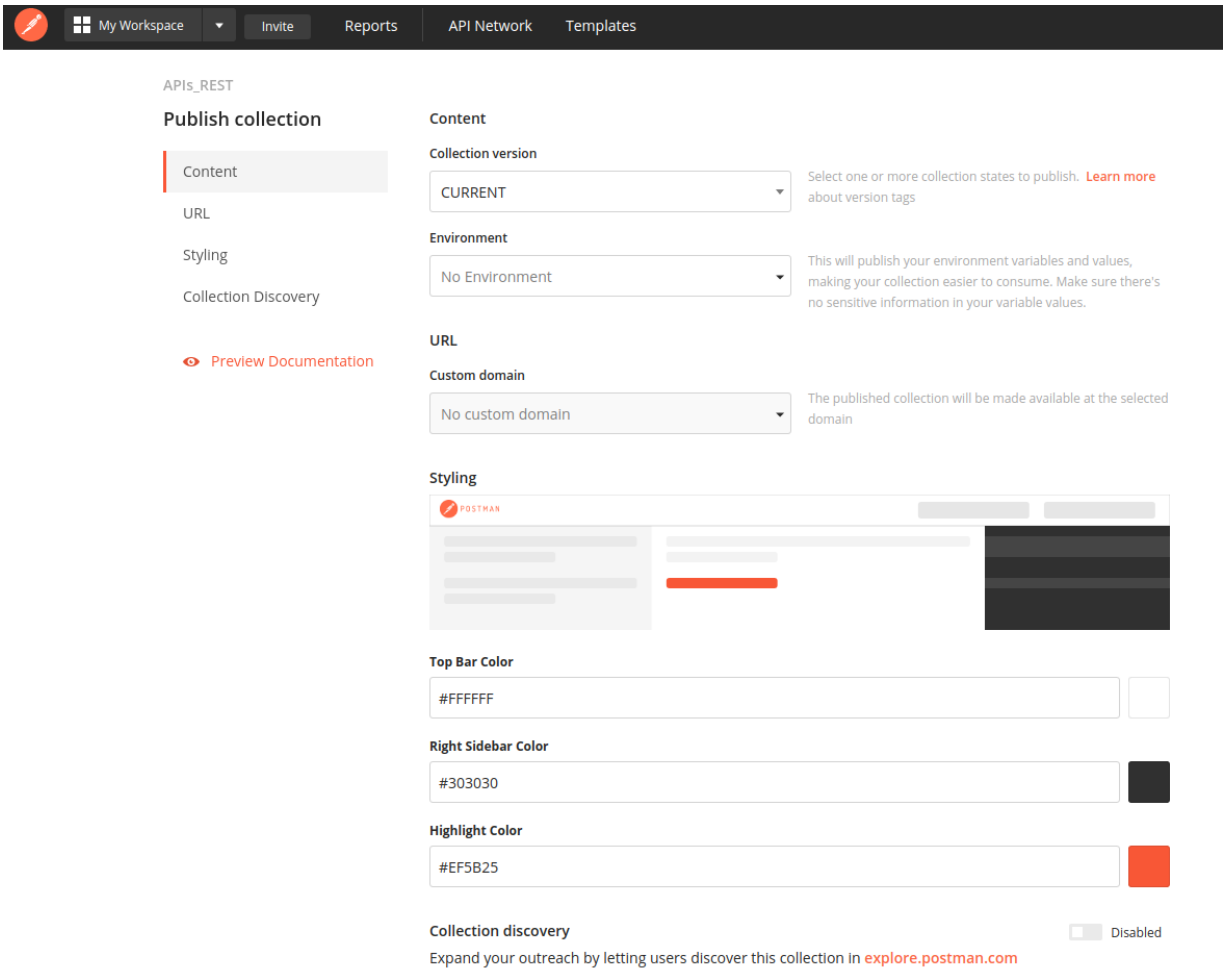


Figura 8.13: Tela de publicação da documentação do Postman

Essa tela permite que se configure (dentre outras coisas) a versão da *collection* a ser utilizada e o *environment*. O *environment* nada mais é do que uma coleção de atributos que pode ser utilizada para alternar de maneira rápida os ambientes que vamos utilizar. Por exemplo, suponha que o *host* dos serviços que utilizamos localmente seja `localhost:8080`, e o de produção `api.car.com.br`.

Então, podemos definir uma variável `host` no *environment* para uma fácil alternância entre esses dois ambientes.

Vamos configurar um *environment*? Para isso, abra novamente o Postman e localize na parte superior à direita da tela um campo assim:

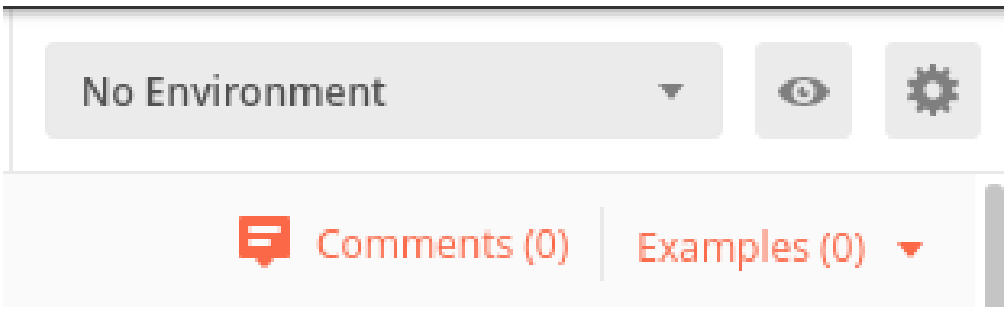


Figura 8.14: Definição do environment do Postman

Clique no ícone da engrenagem. Isso fará com que uma tela de gerenciamento de *environments* se abra:

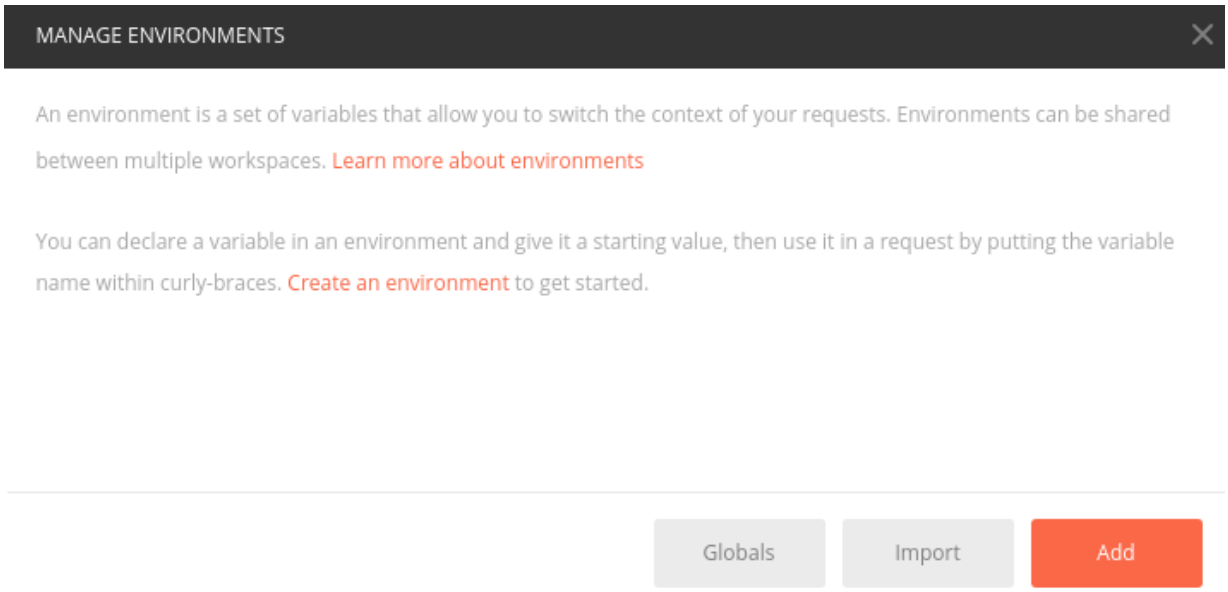

MANAGE ENVIRONMENTS
An environment is a set of variables that allow you to switch the context of your requests. Environments can be shared between multiple workspaces. Learn more about environments
You can declare a variable in an environment and give it a starting value, then use it in a request by putting the variable name within curly-braces. Create an environment to get started.
Globals
Import
Add

Figura 8.15: Tela para criar novos environments

Clique no botão `Add`. Vamos agora criar um ambiente chamado `local-secure` e definir uma variável `host`, com o valor `https://localhost:8080`:

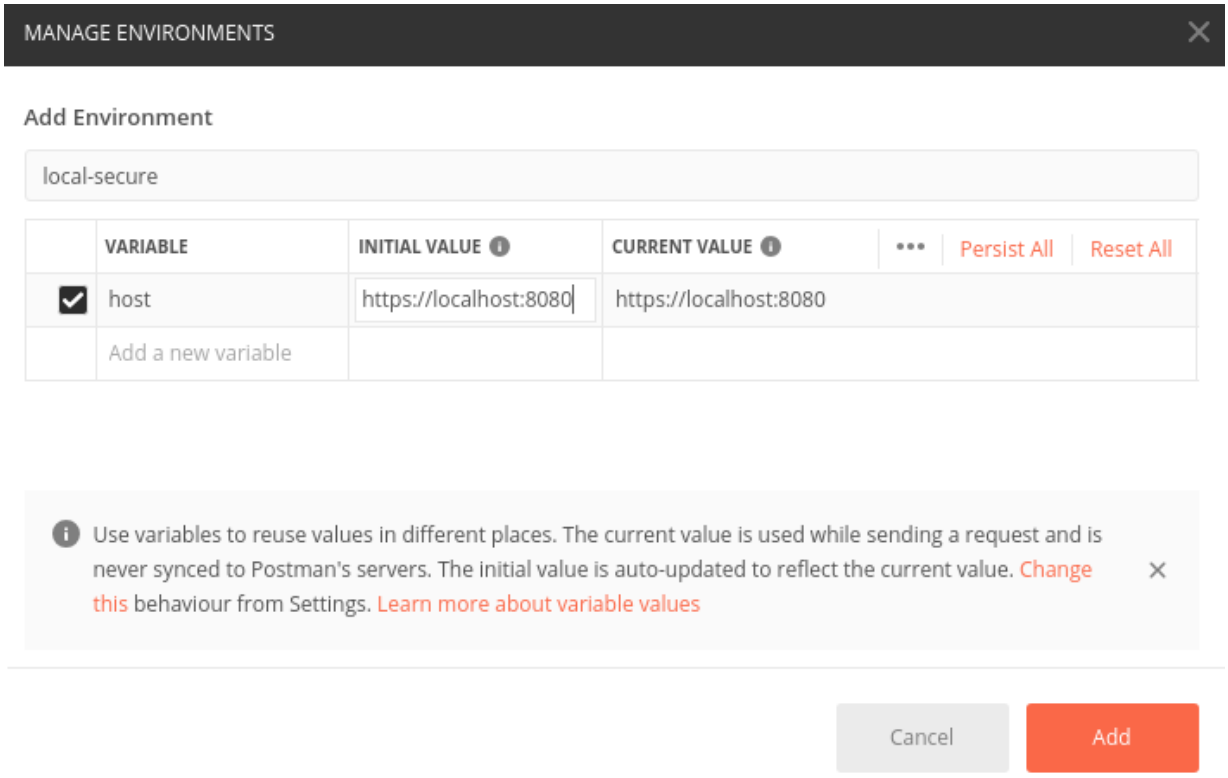

MANAGE ENVIRONMENTS
Add Environment
local-secure
VARIABLE
INITIAL VALUE
CURRENT VALUE
Persist All
Reset All
host
https://localhost:8080
https://localhost:8080
Add a new variable
Use variables to reuse values in different places. The current value is used while sending a request and is never synced to Postman's servers. The initial value is auto-updated to reflect the current value. Change this behaviour from Settings. Learn more about variable values
Cancel
Add

Figura 8.16: Criação do environment local-secure

Para utilizar essa variável, vamos abrir uma das requests, por exemplo, `Listar motoristas` e inserir a variável na URL entre chaves da seguinte forma: `{{host}}/drivers` . Note que o texto da variável fica vermelho se não houver nenhum *environment* no *dropdown*, e laranja se selecionarmos o *environment* `local-secure` , que criamos:

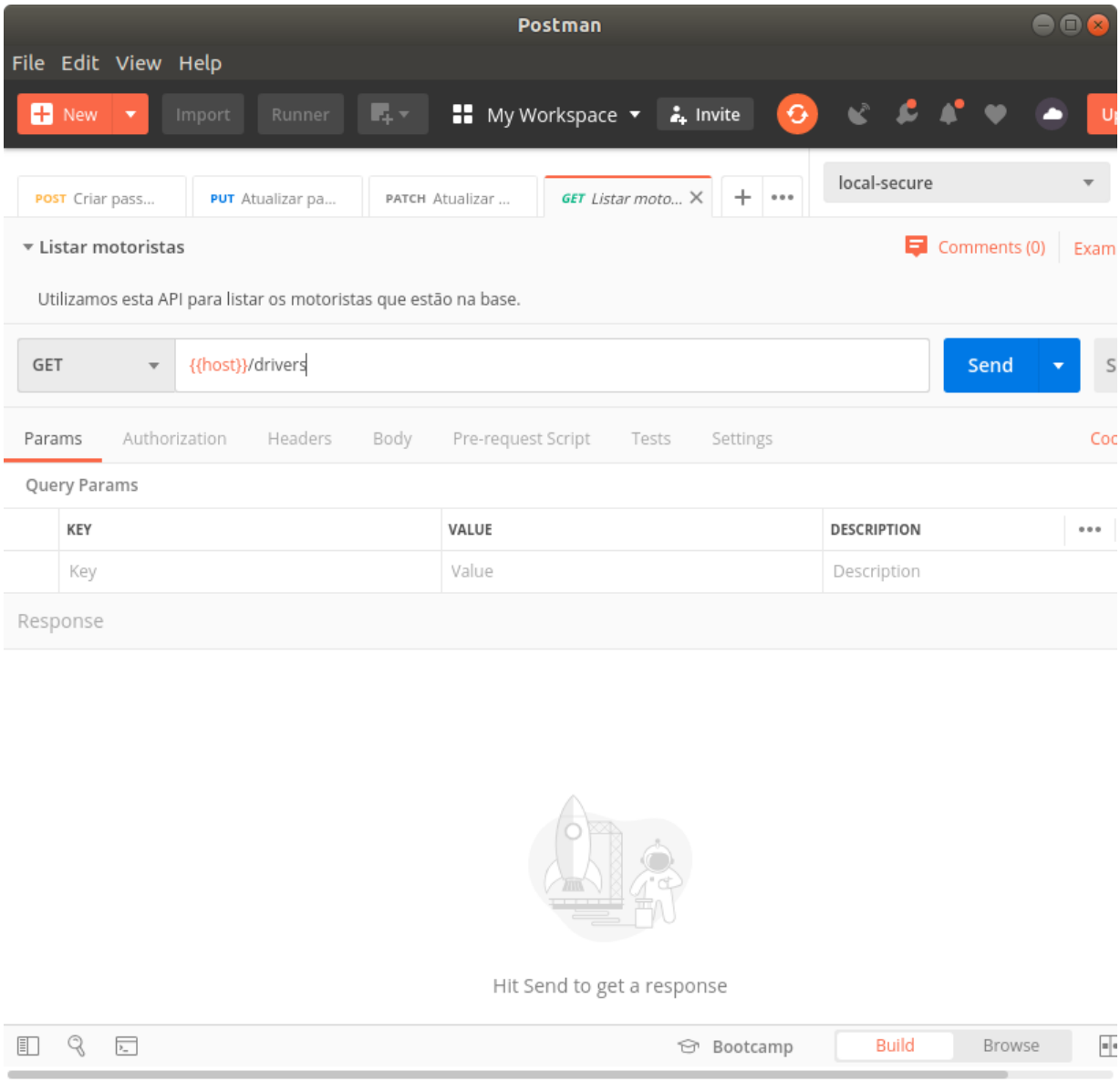


Figura 8.17: Utilizando a variável host

Vamos voltar ao ponto onde estávamos anteriormente clicando no botão `Publish Docs`. Observe que agora aparece o *environment* `local-secure` para publicação. Deixe-o selecionado, como na figura:

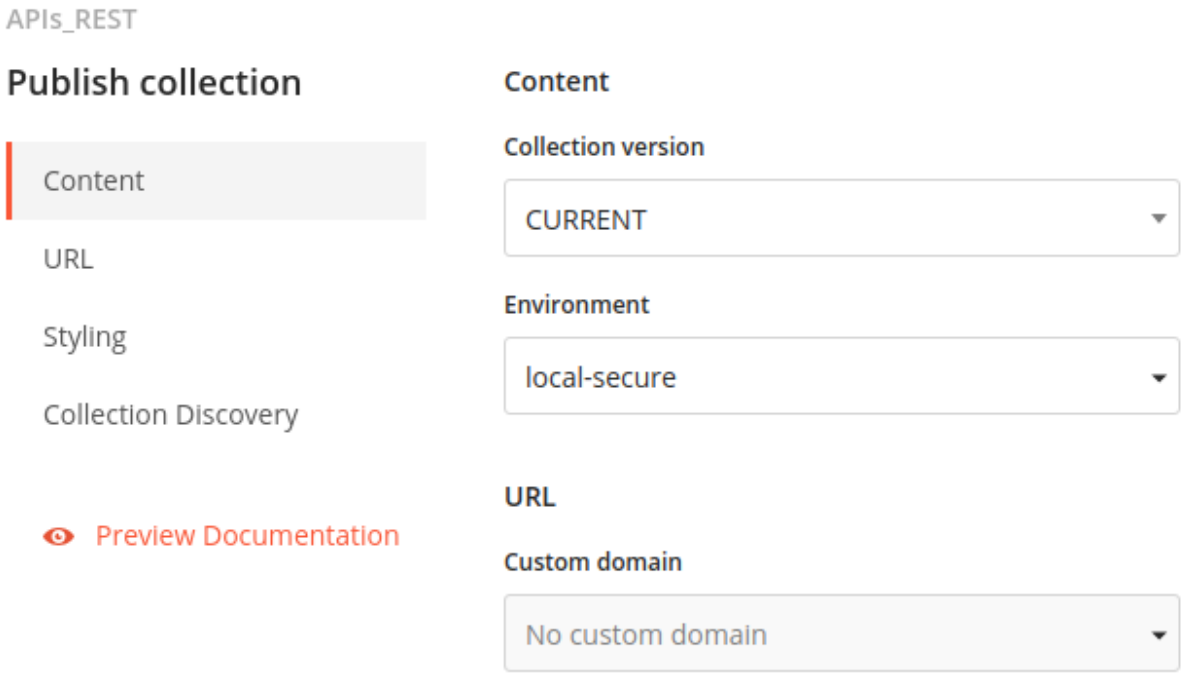


Figura 8.18: Environment local-secure na publicação da documentação

É possível visualizar um *preview* da documentação antes de efetivamente publicá-la, isto é, torná-la publicamente acessível. Para isso, clique no botão `Preview Documentation` . Uma tela semelhante à seguinte se abrirá:

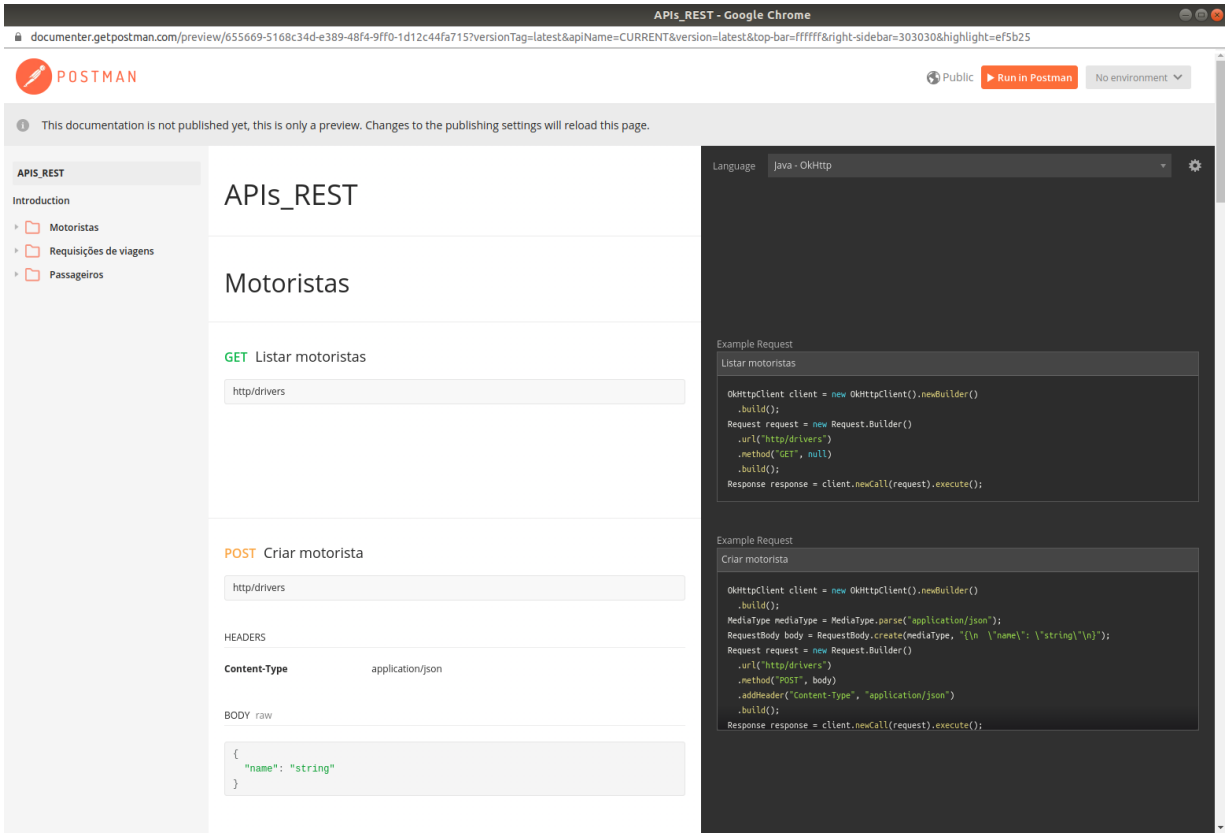


Figura 8.19: Preview da documentação do Postman

Observe que no canto direito da tela estão as *requests* para nossa API. Essa seção disponibiliza um *dropdown* com várias linguagens e frameworks. Trocar essas linguagens fará com que os códigos dos clientes adequados sejam gerados automaticamente, prontos para serem colados na sua IDE e para que você possa trabalhar com essa API diretamente!

Um tipo particular dessas implementações, por exemplo, é o HTTP. Ao selecionar essa opção, você pode visualizar diretamente que tipo de informação é enviada para o servidor:

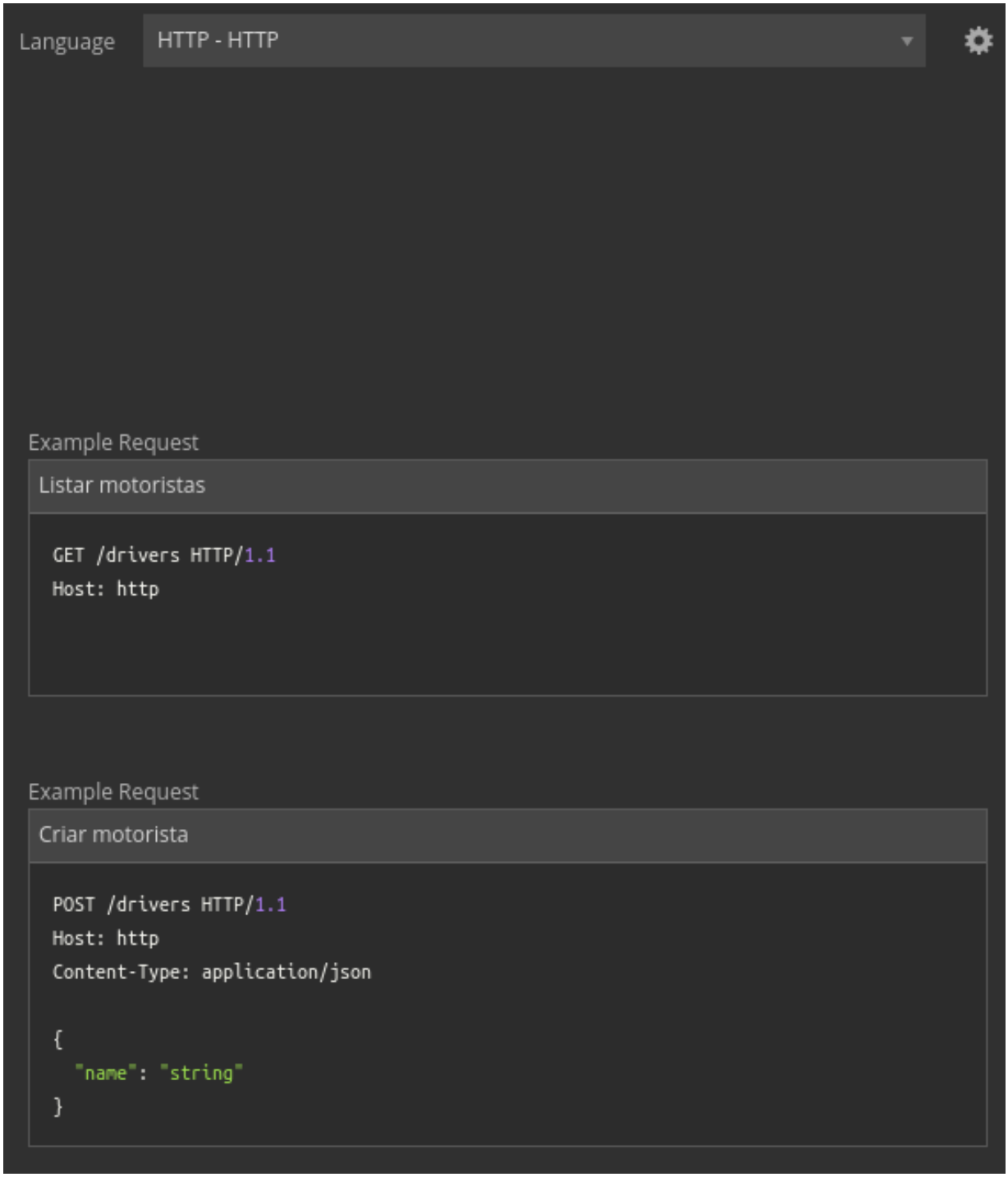


Figura 8.20: Trocando a linguagem do documenter

O último ponto que resta agora é melhorar a documentação. Como exemplo, vamos documentar a API de listar motoristas. Com ela aberta no Postman, clique na seta que aparece à esquerda do

nome. Uma caixa de texto deve se abrir logo abaixo dela, com um botão escrito `Add a description`. Clique nesse botão e insira um texto de documentação, conforme a figura:

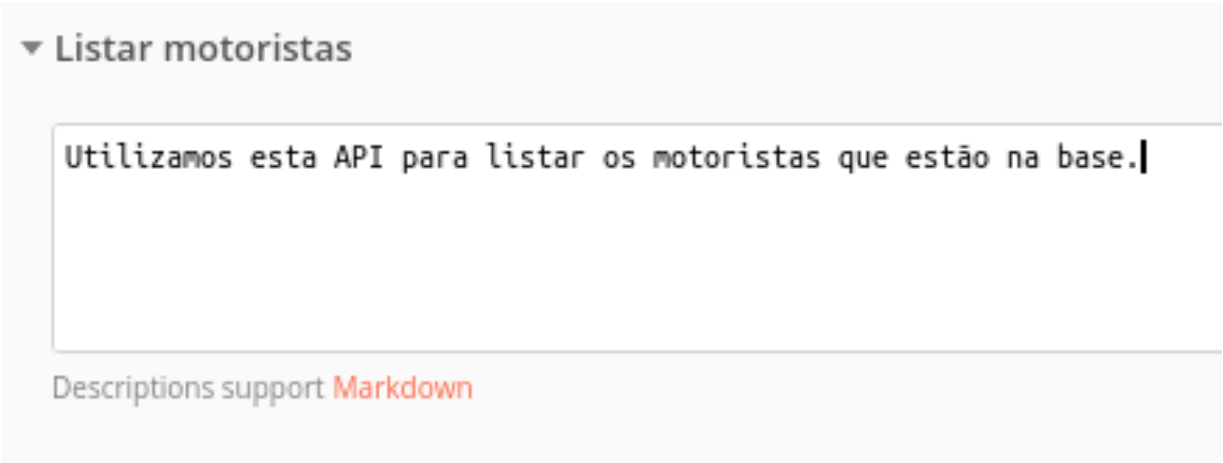


Figura 8.21: Incrementando a documentação

Refaça o processo de publicação e *preview* para ver como ficou:

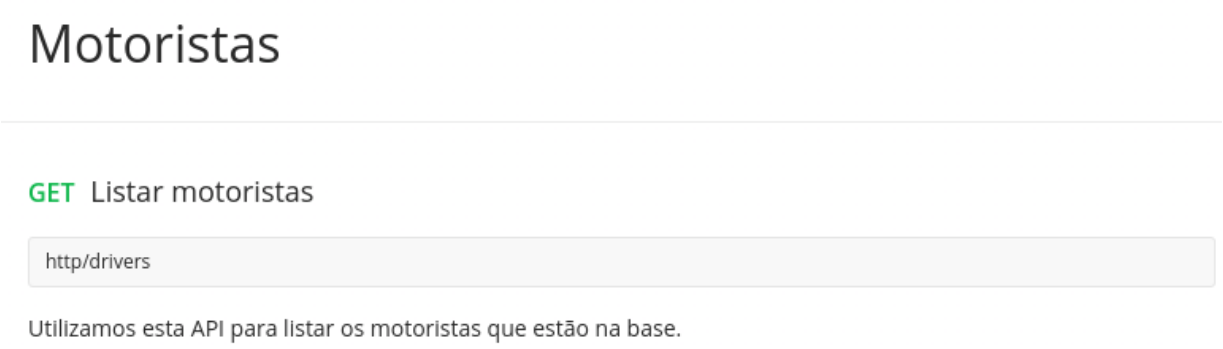


Figura 8.22: Alteração aparecendo no documenter

Quando estiver satisfeito com o resultado, clique no botão `Publish Collection`, que fica no rodapé da página de edição de configurações. A API será publicada e uma tela semelhante à seguinte será vista:

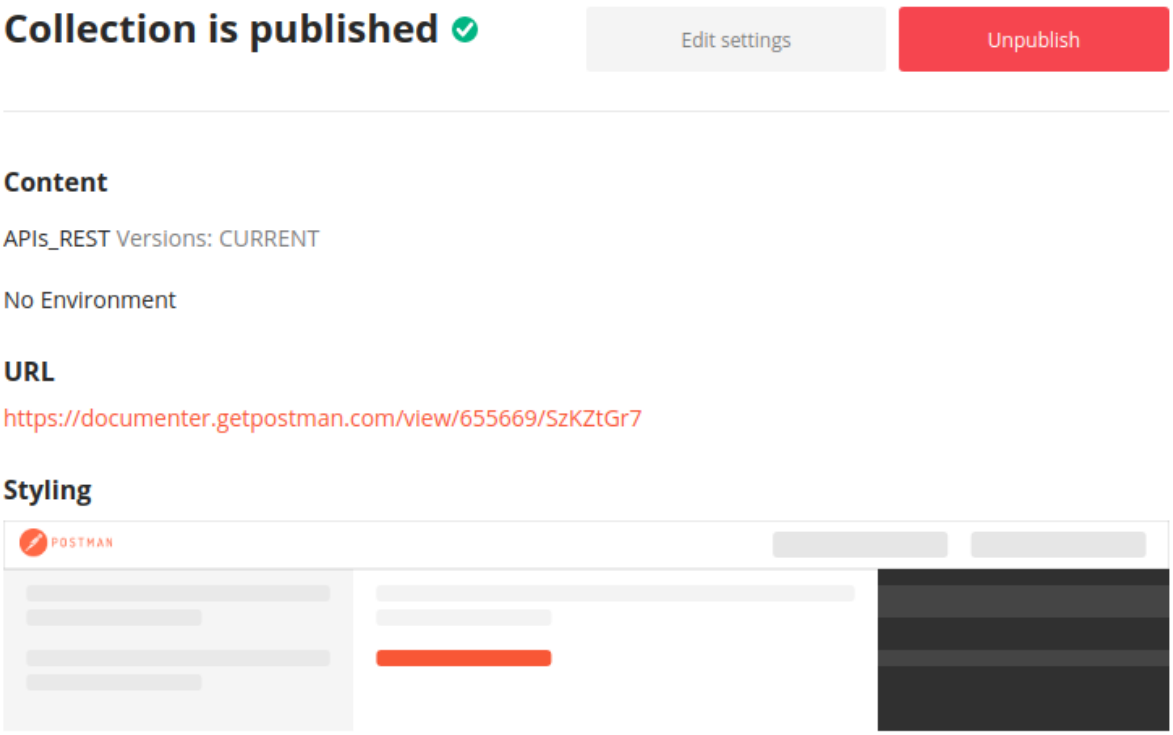


Figura 8.23: Collection publicada

Observe que o Postman gerou uma URL própria para a documentação, que agora está disponível na internet.

> **DICA**
>
> O arquivo fonte dessa coleção está disponível junto com o código-fonte deste livro, em https://github.com/alesaudate/rest-kotlin. Você pode usar o arquivo para publicar diretamente no documenter e testar todos os exemplos deste livro.

Conclusão

Neste capítulo, vimos duas formas diferentes de documentar a sua API. Uma delas é mais próxima de quem desenvolve (o OpenAPI) e a outra, também próxima dos desenvolvedores, busca criar uma

proximidade maior com o pessoal de negócios. Ambas as formas são muito boas de documentar e podem ser usadas concomitantemente, isto é, não é necessário escolher entre uma e outra - você pode usar ambas. Para realizar a escolha, talvez seja interessante ter em mente qual será o público-alvo da documentação da API.

O próprio processo de documentar, entretanto, é extremamente importante. Procure manter isso como uma prática constante, pois fará toda a diferença na evolução dela.

Capítulo 9
Outras técnicas

"A educação é a arma mais poderosa que você pode usar para mudar o mundo." - Nelson Mandela.

Ao longo deste livro, as técnicas foram sendo abordadas conforme criávamos a API C.A.R. No entanto, outras técnicas muito importantes não puderam ser abordadas a tempo, por distintas razões. Como algumas delas são demasiado importantes para ficarem de fora deste livro, este capítulo tem justamente a intenção de abordar um compilado de técnicas para completar o assunto. Para tratar de algumas, vou manter o meu estilo e mostrá-las com exemplos; para outras, vou apenas discorrer a respeito.

9.1 Paginação

Talvez a técnica mais importante que não foi abordada ao longo do livro seja a paginação. Essa é uma técnica extremamente importante para poupar recursos, como memória, CPU, rede etc. Ela consiste em "fatiar" o conteúdo de um determinado recurso e devolver ao cliente da API somente a fatia solicitada. Quando isso é feito, o cliente lida apenas com o conteúdo que realmente importa - se ele tivesse uma grande massa nas mãos imediatamente, fatalmente ele teria que desprezar algo. Isso geraria o desperdício dos recursos já mencionados. O objetivo da paginação é, portanto, minimizar esse desperdício.

Antes de começar, no entanto, deixo um alerta: o Spring é uma ferramenta muito completa e apresenta recursos para realizar várias tarefas de diferentes formas. A forma que apresentarei a seguir foi balanceada para atender aos critérios da interoperabilidade entre os

frameworks utilizados, do modo mais RESTful possível e também para facilitar a implementação de HATEOAS.

Primeiro, analisaremos o que será necessário. A paginação consiste em uma filtragem de recursos, por isso faz sentido que seja aplicada somente sobre a URL de listar vários recursos. Vamos utilizar a API de listagem de motoristas para verificar isso.

Sendo uma filtragem, vamos utilizar *query parameters* para fazer isso. O cliente terá a possibilidade de enviar o número da página desejada. Em um primeiro momento, vamos deixar *hardcoded* o tamanho da página em 20 registros, o que será facilmente substituível.

O primeiro passo é normalizar o nome da URL. Esse é um débito técnico que deixamos quando realizamos a criação da API de motoristas. Para isso, vamos retirar o trecho `/drivers` de todos os mapeamentos da classe e transferi-lo para a anotação `@RequestMapping`, que está na declaração da classe diretamente. Nosso código fica assim:

```
@RequestMapping(path = ["/drivers"], produces =
[MediaType.APPLICATION_JSON_VALUE])
class DriverAPIImpl(
    val driverRepository: DriverRepository
) : DriverAPI{

    @GetMapping
    override fun listDrivers(
        @RequestParam(name = "page", defaultValue = "0") page: Int) =
        //...

    @GetMapping("/{id}")
    override fun findDriver(@PathVariable("id") id: Long) =
        //...

    @PostMapping
    override fun createDriver(@RequestBody driver: Driver) = //...
```

```
    @PutMapping("/{id}")
    override fun fullUpdateDriver(@PathVariable("id") id:Long,
        @RequestBody driver:Driver) : Driver {
        //...
    }

    @PatchMapping("/{id}")
    override fun incrementalUpdateDriver(@PathVariable("id") id:Long,
        @RequestBody driver: PatchDriver) : Driver {
        //...
    }

    @DeleteMapping("/{id}")
    override fun deleteDriver(@PathVariable("id") id: Long) =
        //...
}
```

Agora, vamos alterar o método `listDrivers` para receber como parâmetro o número da página. Como não queremos forçar o cliente a fornecer esse valor, vamos fazer com que ele seja opcional, com valor padrão sendo 0. Criaremos também uma constante para delimitar o número de registros em cada página (no caso, 10):

```
@GetMapping
override fun listDrivers(@RequestParam(name = "page", defaultValue = "0")
page: Int) = //...

companion object {
    private const val PAGE_SIZE: Int = 10
}
```

Para empregar a paginação, vamos utilizar um método presente no nosso repositório. Trata-se do `findAll`, que recebe como parâmetro uma instância de `Pageable`. Podemos criar essa instância utilizando o método `of` da classe `PageRequest`:

```
import org.springframework.data.domain.Page
import org.springframework.data.domain.PageRequest

// ...
```

```
@GetMapping
override fun listDrivers(@RequestParam(name = "page", defaultValue = "0")
page: Int) =
    driverRepository.findAll(PageRequest.of(page, PAGE_SIZE)).content
```

Por si só, a paginação já está pronta. Para testar, vamos subir a aplicação e abrir o Swagger:

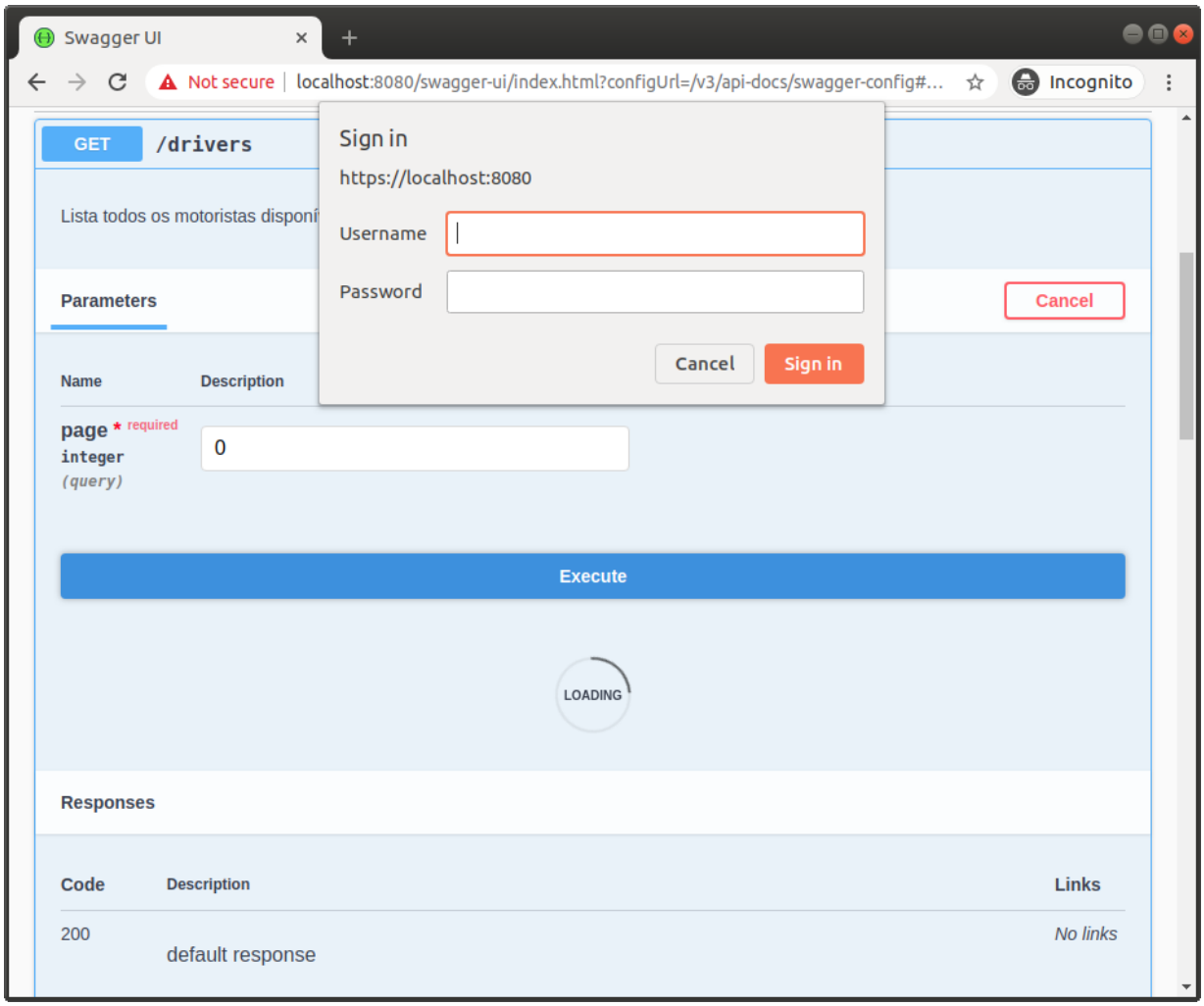


Figura 9.1: Apenas não se esqueça de se autenticar quando testar

Ao receber a resposta de uma primeira requisição, notamos que o resultado vem vazio. Assim sendo, precisamos popular o sistema com dados suficientes (mais de dez motoristas) para que possamos ver a paginação em ação. Uma vez populado, verificamos que,

quando passamos como parâmetro a página 0, um retorno de no máximo dez motoristas vem como resultado. Ao passar como parâmetro a página 1, recebemos um único motorista, ou seja, justamente o que faltava na página 0.

Há um porém. Vamos observar o resultado da página 1:

```
[
  {
    "id": 12,
    "name": "motorista11",
    "birthDate": "2020-03-05T17:18:04.558+0000"
  }
]
```

Observe que não há informações presentes dizendo se esta é a última página, ou qual o endereço da página anterior. Daí, podemos perceber que o uso de HATEOAS traria um grande benefício para essa paginação. Vamos então refatorar a classe `listDrivers` para que esteja adequada aos links HATEOAS.

A primeira coisa a ser feita é trocar a classe de retorno do método para a classe `CollectionModel`. Essa classe é semelhante à `EntityModel` que vimos no capítulo 3, mas especializada para funcionar com coleções. Para utilizá-la, vamos usar o método `of` dela, que recebe uma lista:

```
import org.springframework.hateoas.CollectionModel

// ...

@GetMapping
override fun listDrivers(@RequestParam(name = "page", defaultValue = "0")
page: Int):
        CollectionModel<Driver> {

    val driverPage = driverRepository.findAll(PageRequest.of(page,
PAGE_SIZE))
    val drivers = driverPage.content
```

```
    return CollectionModel.of(drivers)
}
```

O próximo passo é adicionar os links de fato. Para isso, vamos importar diretamente o método `linkTo`, presente no arquivo `WebMvcLinkBuilderDslKt`, para nos ajudar a construir os links que queremos. Para usar esse método, referenciamos um método de uma classe controladora que vai fornecer o link para o cliente. Como temos que referenciar recursivamente o recurso `/drivers`, apenas modificando o valor do *query param* `page`, vamos referenciar a classe `DriverAPIImpl` pelo mecanismo de *generics* do método e, dentro da *closure*, vamos chamar o método `listDrivers`. Para elaborar essa chamada, vamos imaginar com qual parâmetro esse método deve ser chamado para que seja a última página. Não temos acesso direto a esse dado, mas temos como calcular. Acontece que a classe `Page` disponibiliza um método chamado `getTotalPages`, que retorna o número total de páginas. Ou seja, se tivermos uma única página, esse método retornará 1 - que será referente à página de número 0. No nosso caso, como temos duas páginas de dados, a última página é a de número 1 - e o método `getTotalPages` retorna 2. Assim sendo, para calcular qual é a última página, utilizamos o método `getTotalPages` e subtraímos 1.

O código (parcial) fica assim:

```
import org.springframework.hateoas.server.mvc.linkTo
//...

@GetMapping
override fun listDrivers(@RequestParam(name = "page", defaultValue = "0")
page: Int):
        CollectionModel<Driver> {

    val driverPage = driverRepository.findAll(PageRequest.of(page,
PAGE_SIZE))
    val drivers = driverPage.content
    val lastPageLink = linkTo<DriverAPIImpl> {
listDrivers(driverPage.totalPages - 1) }
```

```
    return CollectionModel.of(drivers)
}
```

Por último, vamos definir o relacionamento que esse link possui com o restante do recurso e incluir na resposta. Para isso, vamos usar o método `withRel` do link criado e então, vamos passar o link como parâmetro para o método `of`:

```
@GetMapping
override fun listDrivers(@RequestParam(name = "page", defaultValue = "0")
page: Int):
        CollectionModel<Driver> {

    val driverPage = driverRepository.findAll(PageRequest.of(page,
PAGE_SIZE))
    val drivers = driverPage.content
    val lastPageLink = linkTo<DriverAPIImpl> {
listDrivers(driverPage.totalPages - 1) }
        .withRel("lastPage")
    return CollectionModel.of(drivers, lastPageLink)
}
```

Agora, ao realizar o teste, obtemos algo como:

```
{
  "_embedded": {
    "driverList": [
      {
        "id": 2,
        "name": "motorista1",
        "birthDate": "2020-03-05T18:09:12.017+0000"
      },
      // Vários outros motoristas...
    ]
  },
  "_links": {
    "lastPage": {
      "href": "https://localhost:8080/drivers?page=1"
    }
  }
}
```

Observe a presença do atributo `_embedded` antes da lista de motoristas. Esse é um artifício utilizado pelo Spring HATEOAS para ser aderente à especificação HAL - *Hypertext Application Language* (conforme KELLY, 2013). Trata-se de uma especificação de um formato JSON onde a hipermídia é auto-orientada, de acordo com os preceitos da tese de Roy Fielding. Ao utilizar algumas das *features* do HAL, o usuário seria auto-orientado a seguir os links para prover a funcionalidade de negócio desejada.

O uso do HAL pode trazer vantagens e desvantagens. Enquanto é notório que seguir uma especificação traz benefícios para a padronização e faz com que sua API seja autodescrita e autodocumentada, ela pode trazer complicações no consumo, já que nem todos os clientes podem querer navegar dentro de um atributo com um nome enigmático como `_embedded`. Infelizmente, retirar esse atributo usando a própria API do Spring pode ser um tanto problemático. Ao questionar o time de manutenção do Spring HATEOAS (conforme visto na *thread* do GitHub https://github.com/spring-projects/spring-hateoas/issues/1214 - que por sua vez levou à resposta no Stack Overflow disponível em https://stackoverflow.com/a/60609135/1748132), obtive algumas respostas que mostram apenas que o caminho para remover esse atributo é por meio da criação de sua própria classe de mapeamento.

Assim sendo, vamos criar uma `open class` chamada `Drivers` onde vamos incluir dois atributos: um chamado `drivers`, que vai ser uma lista de `EntityModel<Driver>`, e outro chamado `links`, que vai ser uma lista do tipo `Link`, onde o valor padrão será uma lista vazia. O código dela fica assim:

```
open class Drivers(
    val drivers: List<EntityModel<Driver>>,
    val links: List<Link> = emptyList()
)
```

POR QUE A CLASSE DRIVERS PRECISA SER UMA OPEN CLASS, EM VEZ DE UMA DATA CLASS?

O Spring HATEOAS requer que algumas classes possam ser estendidas para que ele possa instrumentá-las, isto é, incluir comportamentos em tempo de execução. Esse não é o caso das `data classes`, que até o momento não podem ser estendidas. Assim sendo, precisamos criar classes comuns Kotlin e marcá-las com `open` para que o sistema saiba que essas classes podem ser estendidas.

Vamos agora reescrever o código do método `listDrivers` com as mudanças - tanto com o "envelopamento" dos objetos `Driver` dentro de instâncias `EntityModel` quanto com a adoção da classe `Drivers`. O código fica assim:

```
@GetMapping
override fun listDrivers(@RequestParam(name = "page", defaultValue = "0")
page: Int):
        Drivers {
    val driverPage = driverRepository.findAll(PageRequest.of(page,
PAGE_SIZE))
    val drivers = driverPage.content.map { EntityModel.of(it) }
    val lastPageLink = linkTo<DriverAPIImpl> {
listDrivers(driverPage.totalPages - 1) }
        .withRel("lastPage")
    return Drivers(drivers, listOf(lastPageLink))
}
```

Finalmente, obtemos o seguinte resultado:

```
{
   "drivers": [
      {
         "id": 2,
         "name": "string",
         "birthDate": null,
         "links": []
```

```
      },
      {
         "id": 3,
         "name": "string",
         "birthDate": null,
         "links": []
      }
   ],
   "links": [
      {
         "rel": "lastPage",
         "href": "https://localhost:8080/drivers?page=0"
      }
   ]
}
```


9.2 CORS

Uma técnica muito importante que deve ser conhecida por desenvolvedores e desenvolvedoras de APIs é a CORS. Trata-se da sigla para *Cross-Origin Resource Sharing* ou, segundo a tradução da Mozilla disponível em https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Controle_Acesso_CORS, Compartilhamento de recursos com origens diferentes. Trata-se de uma técnica originada a partir de uma necessidade de segurança. Vamos ver como isso funciona:

Suponha que uma API está disponível no servidor `https://api.car.com.br` . Suponha também que uma aplicação JavaScript está sendo executada em `https://bestcars.com` tentando acessar a API. A menos que devidamente instruído, todo *browser* moderno impede esse acesso, por conta de um problema de segurança. Esse problema consiste no seguinte: imagine que `https://bestcars.com` foi hackeado. Os hackers modificaram o código do site para usar a máquina do cliente como um zumbi, e utilizá-la em um ataque DDoS (`Distributed Denial of Service`) sem que o

usuário saiba. Para evitar cenários como esse, foi criada a técnica CORS.

A técnica consiste em realizar a liberação do acesso externo apenas para domínios que são conhecidos. Por exemplo, se `https://bestcars.com` tentar acessar `https://api.car.com.br`, esse acesso será bloqueado a menos que a API da `cars` responda de maneira adequada ao procedimento `CORS`. É uma pequena negociação, que vou explicar com detalhes a seguir.

O cliente JavaScript é instanciado pelo *browser*. Isso faz com que esse cliente já conheça as regras de CORS e faça automaticamente o procedimento. Primeiro, ele envia uma requisição com o método HTTP `OPTIONS`, que tem a função de não retornar nenhum corpo no resultado, somente alguns cabeçalhos de controle que mostram quais métodos HTTP podem ser executados sobre aquela URL. Utilizando a nossa API de motoristas como exemplo, se existir um motorista de ID 1 e o método `OPTIONS` for executado sobre `/drivers/1`, então será informado que os métodos `GET`, `PUT`, `PATCH` e `DELETE` estão disponíveis (além dos métodos `HEAD` e `OPTIONS`).

Se a execução do método `OPTIONS` for parte de uma negociação CORS, o *browser* também enviará na requisição qual a **origem** dela, isto é, a partir de qual domínio está partindo a requisição HTTP. Além disso, o *browser* envia o método que tem a intenção de executar e quais cabeçalhos (se houver cabeçalhos além daqueles que a especificação CORS permite). O servidor responderá com um código de status 200 se a requisição for aceita e 403 caso contrário.

O MÉTODO OPTIONS S É USADO EM CORS?

Em realidade, o método `OPTIONS` foi criado com o intuito de guiar o cliente a respeito da maneira como navegar em uma URL, isto é, apresentando quais métodos HTTP podem ser executados sobre ela. A partir da necessidade de uso da técnica CORS é que o `OPTIONS` evoluiu para tratar essas questões de apresentar outras opções disponíveis.

Para habilitar CORS no nosso projeto, precisamos modificar as configurações do Spring Security. Caso o Spring Security não esteja implementado no seu projeto, consulte a documentação oficial para realizar a configuração.

No método `configure` da classe `SecurityConfig`, logo abaixo da declaração onde estamos desabilitando o CSRF, vamos incluir uma chamada ao método `cors()`. Esse método vai inicializar a configuração para nós:

```
override fun configure(http: HttpSecurity) {
    http.csrf().disable()
    http.cors()
   // restante do método
}
```

Vamos também incluir um *bean* chamado `CorsConfigurationSource`. Esse bean é utilizado para "amarrar" a configuração do CORS da aplicação com as URLs sobre as quais a configuração é válida. Para declarar de quais origens métodos HTTP e cabeçalhos são aceitos, utilizamos a classe `CorsConfiguration`. Para realizar esse controle, utilizamos os métodos `setAllowedOrigins`, `setAllowedMethods` e `addAllowedHeader`, respectivamente. A declaração é feita assim:

```
import org.springframework.web.cors.CorsConfiguration
import org.springframework.web.cors.CorsConfigurationSource

@Bean
```

```
fun corsConfigurationSource(): CorsConfigurationSource {
    val configuration = CorsConfiguration()
    configuration.allowedOrigins = listOf("https://bestcars.com")
    configuration.allowedMethods =
listOf("GET","POST","PUT","DELETE","PATCH")
    configuration.addAllowedHeader("*")

    // ...
}
```

Note que, ao utilizar `*` no método `addAllowedHeader`, estamos declarando que todos os cabeçalhos são permitidos. Isso também é suportado nas outras declarações (ou seja, indicando que qualquer origem de dados e qualquer método HTTP são permitidos), embora não seja recomendado, já que o CORS existe por motivos de segurança.

Finalmente, para declarar quais URLs da nossa API estarão com essa configuração habilitada, vamos utilizar a classe `UrlBasedCorsConfigurationSource`. Essa classe vincula uma URL, ou padrão de URL, a uma configuração CORS. Para realizar esse vínculo, utilizamos o método `registerCorsConfiguration`, passando como parâmetro o padrão `/**` (que vai interceptar todas as URLs) e a configuração que criamos anteriormente. Vale observar que a classe `UrlBasedCorsConfigurationSource` implementa a interface `CorsConfigurationSource`, ou seja, podemos retornar essa implementação no nosso método:

```
import org.springframework.web.cors.CorsConfiguration
import org.springframework.web.cors.CorsConfigurationSource
import org.springframework.web.cors.UrlBasedCorsConfigurationSource

@Bean
fun corsConfigurationSource(): CorsConfigurationSource {
    val configuration = CorsConfiguration()
    configuration.allowedOrigins = listOf("https://bestcars.com")
    configuration.allowedMethods =
listOf("GET","POST","PUT","DELETE","PATCH")
    configuration.addAllowedHeader("*")
```

```
    val source = UrlBasedCorsConfigurationSource()
    source.registerCorsConfiguration("/**", configuration)
    return source
}
```

Agora vamos testar como ficou a nossa configuração. Utilizaremos o método POST em `/drivers` para verificar o funcionamento do CORS no nosso ambiente e fazer, inicialmente, três testes. O primeiro verificará como a API se comporta sem que utilizemos qualquer um dos recursos de que CORS necessita para funcionar. No segundo, vamos utilizar o cabeçalho `Origin` para informar ao servidor qual é a origem de dados e verificar qual é a resposta, mas informaremos uma origem que não está na configuração que criamos. No último teste, vamos informar a origem `https://bestcars.com` , que foi a registrada na configuração do CORS, e verificar como o sistema se comporta.

Vamos ao primeiro teste:

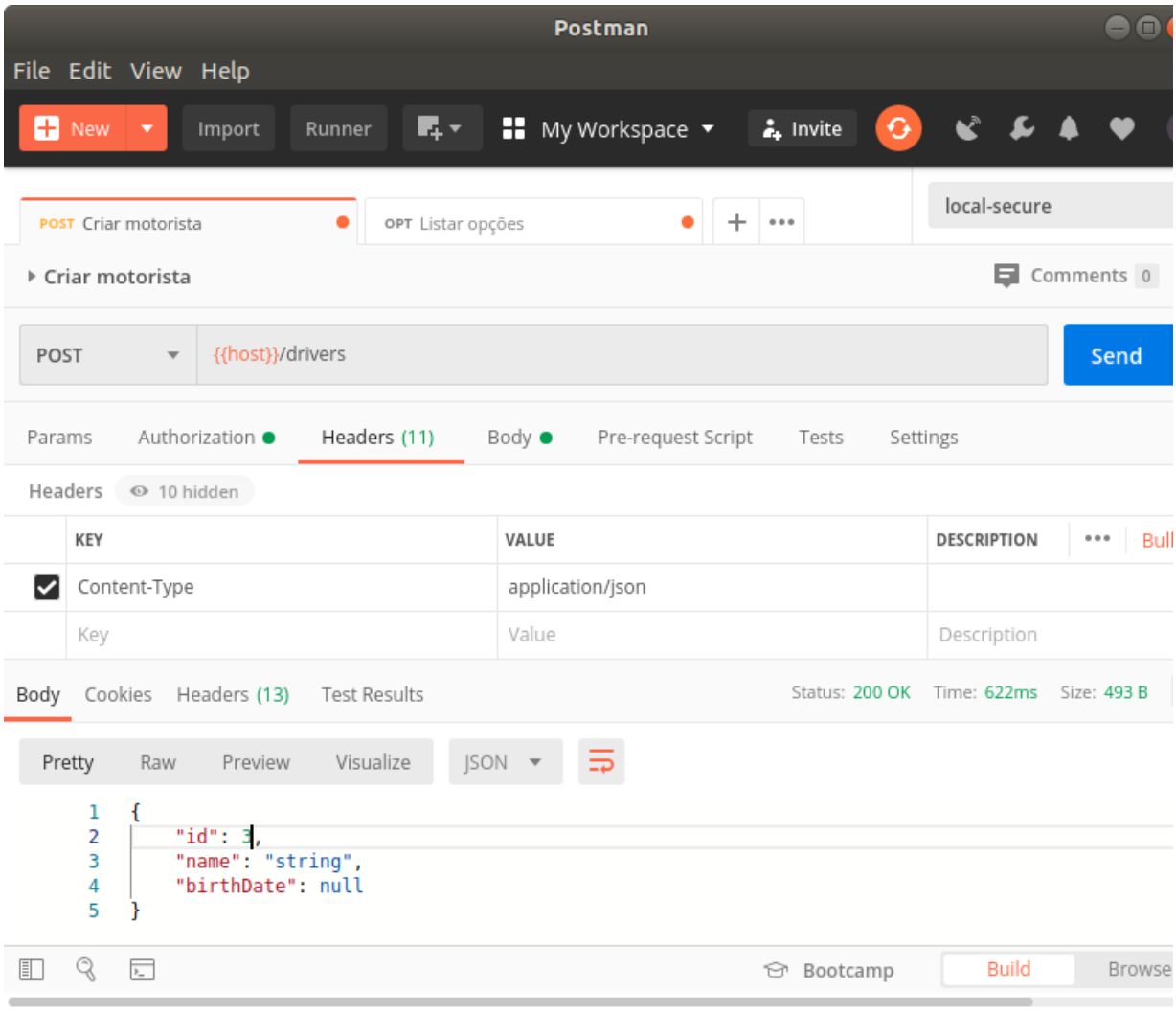


Figura 9.2: Requisição sem o cabeçalho Origin

Como se pode observar, a API se comporta da maneira normal. Esse comportamento tem como público-alvo clientes que não precisam de CORS, ou seja, que não são aplicações JavaScript. Dessa forma, a API continua se comportando como se não houvesse configuração de CORS ajustada. Um lembrete: o próprio *browser* ajusta o cabeçalho `Origin` nas requisições informando a origem.

Se fizermos a mesma requisição, mas ajustando o cabeçalho `Origin` com um domínio desconhecido do servidor, recebemos uma

resposta com o código 403 e um texto dizendo que a requisição é inválida:

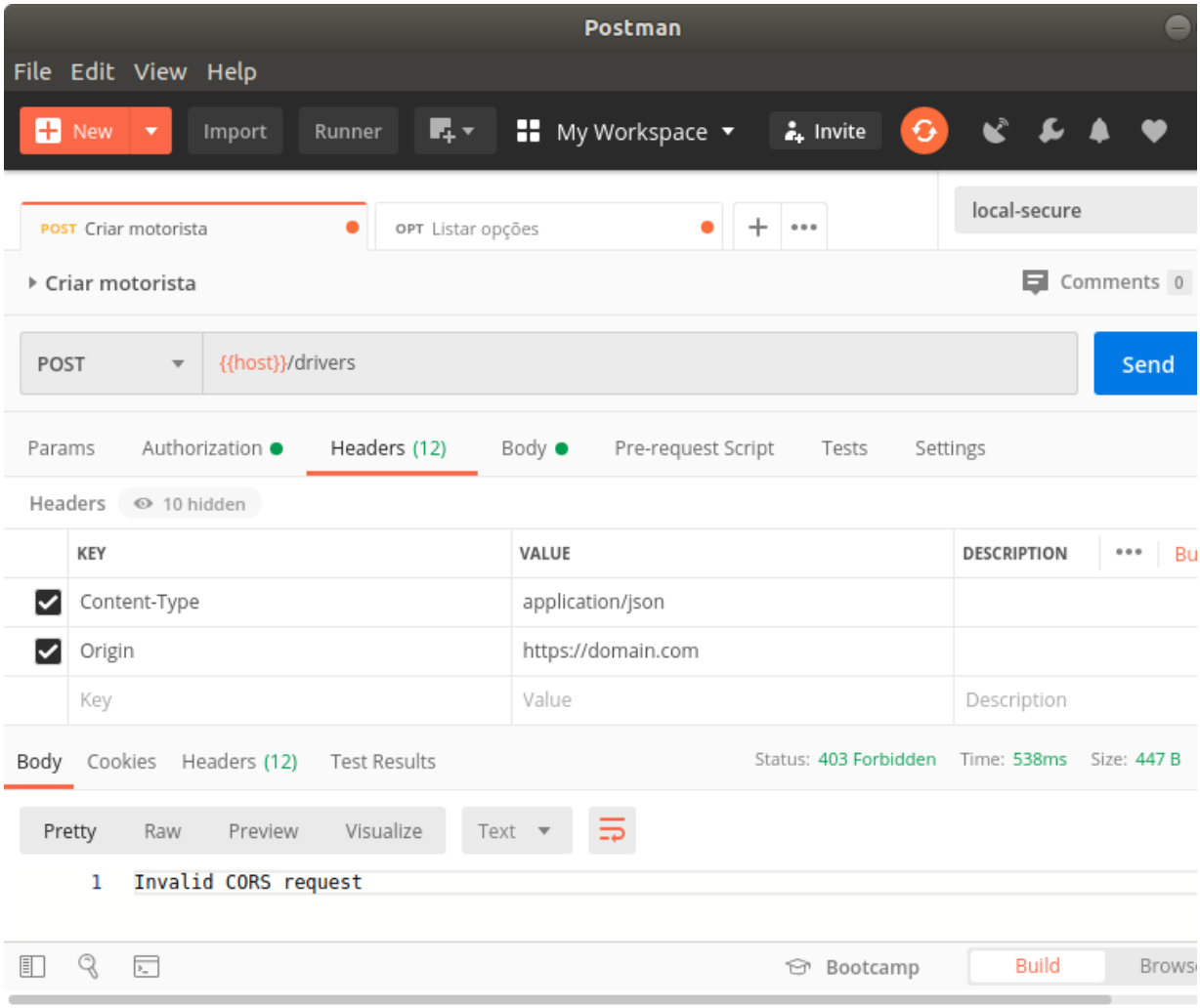


Figura 9.3: Requisição com o cabeçalho Origin numa origem não cadastrada

Vamos então realizar a requisição com o cabeçalho `Origin` ajustado para `https://bestcars.com`, como foi configurado:

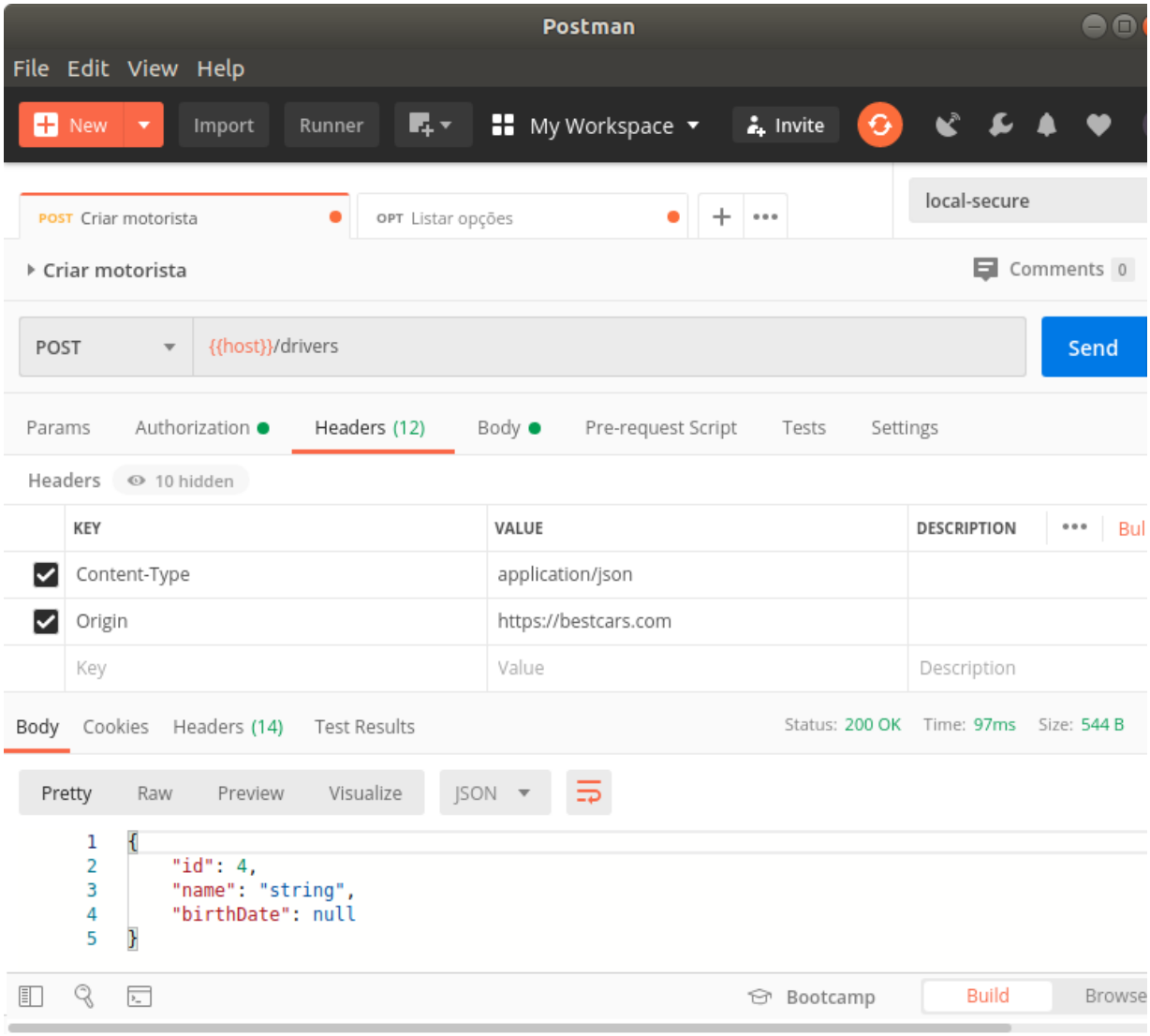


Figura 9.4: Requisição com o cabeçalho Origin ajustado em https://bestcars.com

Vemos que tudo transcorre normalmente dessa forma. Temos certeza de que nossa configuração CORS está se comportando da maneira esperada e rejeitando corretamente requisições inválidas. Mas, quando as requisições são feitas por clientes JavaScript hospedados em *browsers*, existe um passo que deve ser executado anteriormente. Esse passo consiste em verificar se a origem dos dados é aceita e quais métodos HTTP e cabeçalhos são suportados pela API. O servidor retorna esses dados como uma lista, de forma que o cliente pode deixar o resultado armazenado em *cache* e não realizar a mesma requisição várias vezes.

Essa requisição consiste no uso do método HTTP `OPTIONS`, conforme explicado anteriormente. Quando executado fora de um contexto CORS, esse método tem a função de orientar o cliente sobre quais métodos HTTP estão disponíveis para execução na URL informada. No contexto CORS, os cabeçalhos `Origin`, `Access-Control-Request-Method` e `Access-Control-Request-Headers` também são utilizados para informar, respectivamente, se a origem informada está disponível, além de quais métodos HTTP e cabeçalhos.

> **POR QUE UMA REQUISIÇÃO É ENVIADA ANTES DA REQUISIÇÃO "DE VERDADE"? ISSO NÃO É UM DESPERDÍCIO DE RECURSOS?**
>
> A requisição que é feita pelo CORS usando o método `OPTIONS` é chamada de requisição *preflight*. Ela não é enviada sempre pelo *browser* - existem condições específicas em que o *browser* considera ser seguro enviar a requisição para a API sem realizar essa negociação. Não pretendo listar aqui quais são essas condições, mas posso resumi-las como condições que remetem à navegação em páginas web, e não ao consumo de APIs REST. Quando estamos falando de uma API REST, o *browser* prefere mandar o *preflight* por ser uma forma de, teoricamente, utilizar um método que não gera quaisquer efeitos colaterais antes de realizar a requisição pretendida - que, por sua vez, pode gerar. Dessa forma, o *preflight* é necessário como um meio de proteção contra efeitos colaterais.

Vamos verificar isso em funcionamento. Execute o método `OPTIONS` sobre a URL `/drivers`, passando o cabeçalho `Origin` com `https://bestcars.com`, `Access-Control-Request-Method` com `POST` e `Access-Control-Request-Headers` com um cabeçalho qualquer (utilizei no exemplo `x-custom-header`). Observe que, como configuramos `*` no servidor para os cabeçalhos, então todos são aceitos.

Essa requisição retorna da seguinte forma:

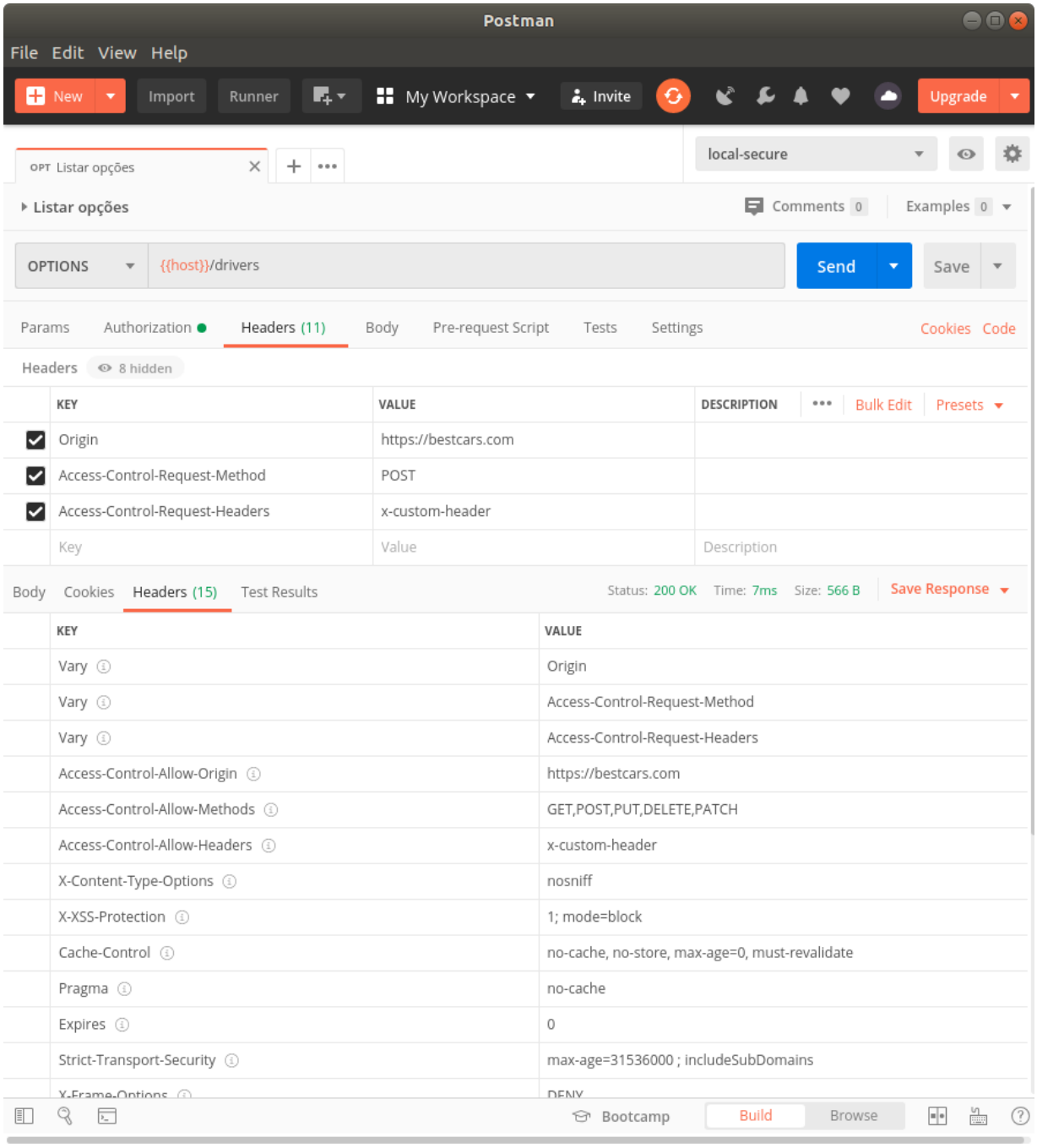


Figura 9.5: Execução do método OPTIONS com todos os valores válidos

Observe que, dentre os muitos cabeçalhos retornados, temos o `Access-Control-Allow-Methods` com `GET,POST,PUT,DELETE,PATCH` e o `Access-Control-Allow-Headers` com `x-custom-header` (o servidor não informa que todos os cabeçalhos são aceitos). Além disso, o código de

status 200 indica que a requisição `POST` com o cabeçalho citado pode ser feita a partir dessa origem.

Vamos checar o que acontece se trocarmos a origem:

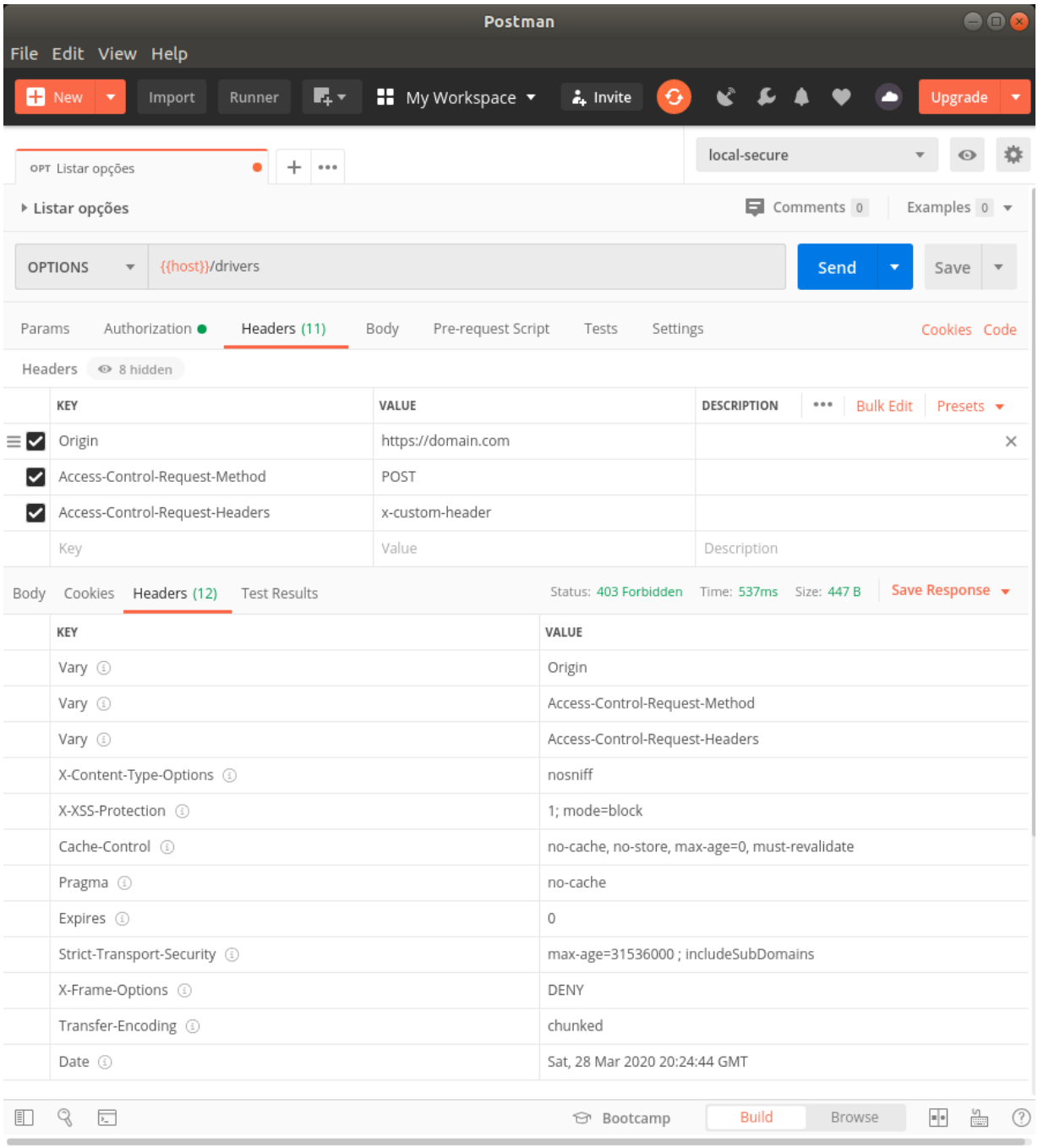


Figura 9.6: Execução do método OPTIONS com uma origem inválida

Da mesma forma que ocorreu com o método `POST` executado, o código de status também mudou para 403, ou seja, essa requisição não será aceita.

Por último, vamos voltar a origem para `https://bestcars.com` e informar um método que não é aceito - digamos, o método `HEAD` :

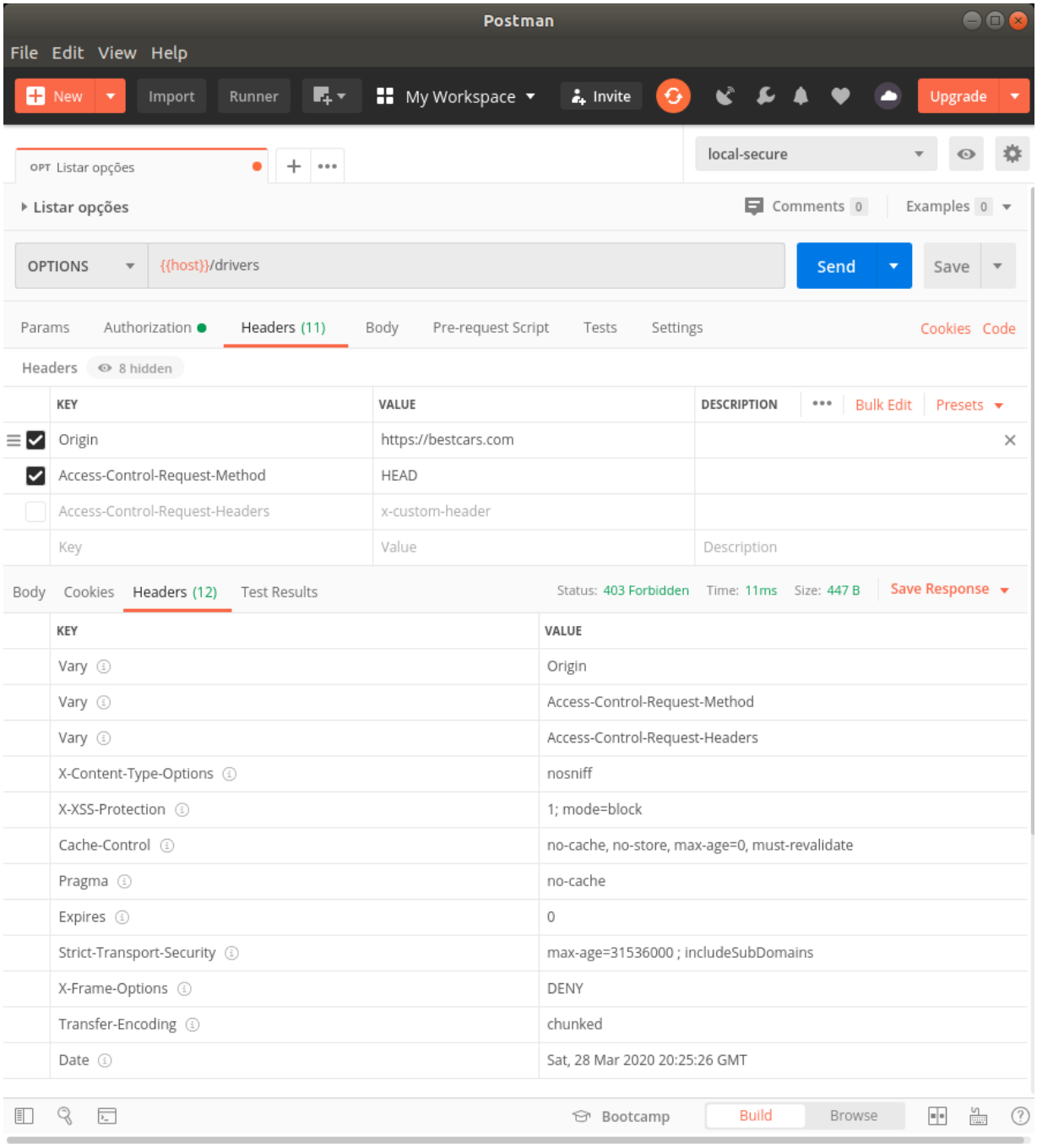


Figura 9.7: Execução do método OPTIONS com um método HTTP inválido

Mais uma vez, recebemos o código de status 403.

9.3 OAuth

OAuth é a especificação de uma técnica de autenticação que permite que um site trabalhe com dados que venham de outro site - sem que seja necessário fornecer a senha do usuário final. Isso abre possibilidades para operações que hoje são bastante comuns, como realizar o cadastro em um site utilizando as credenciais de uma rede social, por exemplo.

Quando a especificação surgiu (OAuth 1), utilizou-se na época uma analogia com carros de luxo e o `modo valet` (presente no Chevrolet Corvette C4 ZR-1, por exemplo). Funciona da seguinte forma: depois de criar várias APIs para o sistema da sua startup, você decide comprar um carro de luxo, muito potente, com cerca de 700 cv de motor. Então você decide pegar esse carro e ir a um restaurante badalado da cidade, onde o manobrista vai recebê-lo. E é aí que você se depara com a cena: Você. Deixando. A chave. Na mão. Do manobrista. Que você não conhece. Cujas intenções você não conhece. Cena aterradora de se pensar. Aliás, se você assistiu *Curtindo a Vida Adoidado*, você já pensou.

Daí vem a ideia de se criar uma *segunda* chave para o carro. Uma que, quando utilizada, corta a potência do motor pela metade, não permite que se abra o porta-malas e dá autonomia para que se ande apenas por poucos quilômetros com o carro.

O protocolo OAuth seguia com essa mesma noção. O carro são seus dados. Você continua sendo você. Seu usuário e senha são a chave para acesso aos dados. E o site interessado em acessar seus dados é o manobrista. O protocolo OAuth implanta a noção de se ter um token especial para que seja possível que o site acesse somente aquilo de que ele necessita sem ter que fornecer o usuário e senha. Como "bônus" esse token é revogável, ou seja, você tem total controle do que vai ser acessado pelo site e pode remover esse acesso quando achar melhor.

A versão atual do OAuth é a 2. Algumas coisas mudaram entre as especificações, sendo que elas não são retrocompatíveis e até

mesmo algumas nomenclaturas foram modificadas. Para conhecer o fluxo, vamos conhecer a nomenclatura dos envolvidos no processo:

- **Dono do recurso** - Geralmente, é você.
- **Cliente** - É a aplicação interessada em acessar o recurso.
- **Servidor de recursos** - É a aplicação onde o recurso (ou dados) está hospedado.
- **Servidor de autorização** - É o servidor que cria a verificação de acordo com o protocolo OAuth. Na maioria das vezes, está junto do servidor de recursos.

Para que um fluxo do OAuth funcione, uma série de etapas precisam ser concluídas. Para simplificar, vamos imaginar que você deseja disponibilizar para os usuários do sistema C.A.R. um *login* utilizando dados de uma rede social qualquer (digamos, alguma que tenha um logotipo azul e branco). Estas etapas, de forma simplificada, são as seguintes:

1. Ao clicar no botão de *login* do C.A.R., o usuário manifesta seu desejo de utilizar seus dados da rede social;
2. O C.A.R., que já estava cadastrado na rede social e, portanto, possui um `client ID` e um `client secret`, notifica a rede social desse desejo e apresenta seus dados para a rede social;
3. A rede social devolve um *link* para que o usuário seja redirecionado;
4. O usuário, ao ser redirecionado, se depara com uma tela informando o uso que o C.A.R. fará de seus dados. Ao concordar, a rede social redireciona o usuário para o C.A.R. com um código de acesso;
5. De posse desse código de acesso, o C.A.R. faz uma nova requisição para a rede social, solicitando o **access token**, que permite que o C.A.R. acesse o conjunto de dados que foi autorizado pelo cliente.

Faço uma observação importante aqui: **o fluxo mencionado acima precisa de um browser**. E ele é apenas um dos fluxos disponíveis no OAuth 2. Na época do OAuth 1, havia só um fluxo possível

(similar ao descrito acima, porém com alguns detalhes técnicos implícitos). O OAuth 2 veio para trazer mais pragmatismo à especificação e acrescentar vários fluxos diferentes de acordo com a necessidade, tornando possível inclusive que APIs REST usem OAuth sem *browser*. Alguns dos fluxos disponíveis são os seguintes:

- *Authorization Code Flow*
- *Client Credentials Flow*
- *Device Flow*
- *Password Credentials Flow*

Vamos revisá-los e verificar quando usar cada um.

Authorization Code Flow

Este fluxo é o mais semelhante ao OAuth 1 e é representado pelo passo a passo dado acima. Foi criado com a intenção de ser utilizado quando o cliente é uma **aplicação web**, em que o usuário utiliza um *browser* para realizar o acesso.

Quando o usuário utiliza alguma espécie de recurso que ativa a necessidade do OAuth, o cliente o redireciona para o servidor de autorização para realizar o processo de consentimento, que consiste em revisar uma lista de informações às quais o cliente terá acesso e um formulário de *login*. Quando o usuário faz *login*, ele está dando o consentimento para que o cliente execute as ações que precisa sobre os dados listados.

Então, o servidor de autorização redirecionará o usuário para o sistema do cliente, portando um código de autorização (ou *authorization code*, que é o nome do fluxo). O cliente recebe esse código de autorização e o troca no servidor de autorização por um *access token*. Esse *token* é utilizado em todas as chamadas que serão feitas para o servidor de recursos, que os fornecerá, pois o *token* concede esse acesso ao cliente.

Client Credentials Flow

Este fluxo é mais semelhante a um fluxo de usuário/senha convencional, com a diferença de que tem seu público-alvo em aplicações. Ou seja, o seu uso tem a intenção de permitir que uma aplicação faça *login* em outra aplicação, sem necessariamente utilizar usuário e senha de uma pessoa ou alguma senha "inventada". Seu uso não é muito diferente de um fluxo intuitivo de usuário e senha:

1. O cliente (aplicação) envia um par `client ID` e `client secret` para o servidor de autorização;
2. O servidor de autorização valida e, caso estejam corretos, retorna um `access token` para o cliente;
3. O cliente então vai até o servidor de recursos utilizando o `access token` e tem acesso aos seus dados.

Device Flow

Este fluxo procura focar em dispositivos que tenham acesso limitado à internet e, portanto, talvez ofereçam uma experiência de uso pobre para seus usuários. Focando nesse detalhe, o fluxo procura oferecer um link para que o usuário consiga continuar de onde parou.

Por exemplo: ele inicia o processo em um celular, continua do computador e, quando volta para o celular, todo o fluxo está finalizado. Esse fluxo, portanto, tem três estágios, cujas etapas são as seguintes:

1. O usuário utiliza seu dispositivo móvel para requisitar os dados;
2. O dispositivo envia uma requisição para o servidor de autorização;
3. O servidor retorna o `Device Code` (ID do dispositivo registrado no servidor), o `User Code` (ID do cliente registrado) e a `Verification URL` (URL de onde o fluxo poderá prosseguir);

4. O dispositivo apresenta o `User Code` e a `Verification URL` para o cliente;
5. Já no computador, o usuário utiliza um *browser* para acessar a URL de verificação que contém seu código;
6. O *browser* apresenta a tela de consentimento;
7. Quando o usuário dá seu consentimento, o servidor de autorização marca o dispositivo como autorizado;
8. Enquanto isso, o dispositivo móvel fica fazendo *polling* no servidor para verificar se já há uma resposta;
9. Quando o dispositivo é marcado como autorizado, a próxima requisição *polling* que o dispositivo fará vai obter o `access token`, permitindo que o fluxo continue pelo dispositivo móvel.

Password Credentials Flow

O fluxo de `password credentials` é um caso único dentre os fluxos de OAuth. Ele pressupõe confiança total no cliente, a ponto de o usuário fornecer o usuário e senha reais da aplicação terceira (ou seja, na realidade ele ignora aquela explicação do carro, `valey key` etc.). Esses casos são aqueles em que o cliente é o Sistema Operacional do computador, por exemplo. Mas também claramente é uma oferta para que os clientes iniciem migração de fluxos onde se utiliza HTTP Basic para autenticação utilizando OAuth.

O fluxo funciona da seguinte forma:

1. O usuário apresenta suas credenciais (usuário e senha) para o cliente, expressando seu desejo de utilizar os recursos do servidor de recursos;
2. O cliente apresenta essas mesmas credenciais para o servidor de autorização;
3. O servidor de autorização realiza a autenticação dos dados e devolve o `access token`;
4. O cliente utiliza o `access token` para acessar os dados.

Existem também outros fluxos OAuth, bem como variações desses apresentados aqui. Em realidade, seria necessário um livro inteiro

apenas para falar sobre OAuth, o que de fato existe e está referenciado no próximo capítulo. Caso você se interesse pelo assunto, ele será um ótimo ponto de partida para se aprofundar mais.

9.4 AWS API Gateway

Eu não poderia escrever um livro sobre APIs sem falar de *API Gateways*. Essa é uma classe de sistemas cuja missão é principalmente monitorar APIs e retirar delas preocupações ortogonais, deixando detalhes de implementação apenas nos serviços. Isso inclui preocupações com autenticação e autorização, limitação de capacidade de uso, roteamento entre diferentes provedores do serviço etc.

Existem vários *API Gateways* disponíveis no mercado. Um deles é o Kong (disponível em https://konghq.com/kong/), que é *open source* e pode ser instalado e gerenciado manualmente. No entanto, darei preferência aqui ao *API Gateway* da AWS, uma vez que a AWS é, no momento, um dos *cloud providers* mais utilizados no mundo.

O console da AWS está disponível em https://aws.amazon.com/pt/console/. Você pode acessar o console e criar uma conta usando seu cartão de crédito.

AVISO: enquanto existem muitas ferramentas AWS que você pode usar gratuitamente, algumas são pagas. No momento da escrita deste livro, o AWS API Gateway é **gratuito** para até 1 milhão de chamadas da API por mês, conforme a seguir:

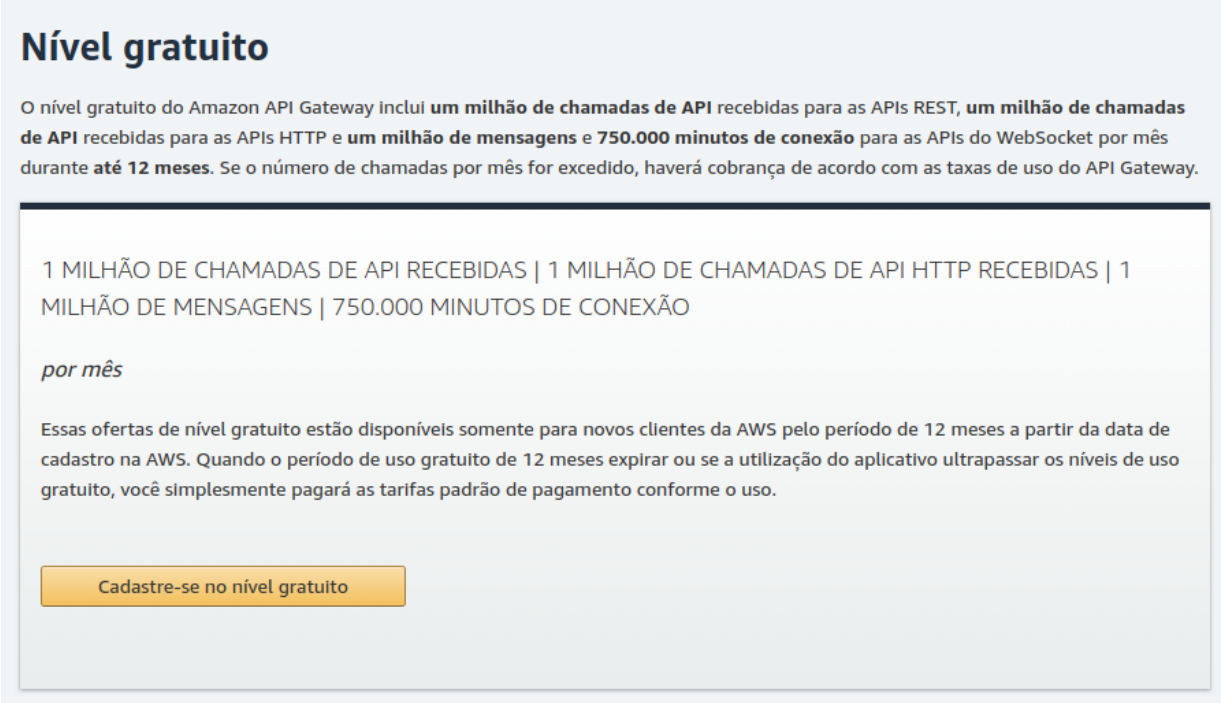

Nível gratuito
O nível gratuito do Amazon API Gateway inclui um milhão de chamadas de API recebidas para as APIs REST, um milhão de chamadas de API recebidas para as APIs HTTP e um milhão de mensagens e 750.000 minutos de conexão para as APIs do WebSocket por mês durante até 12 meses. Se o número de chamadas por mês for excedido, haverá cobrança de acordo com as taxas de uso do API Gateway.
1 MILHÃO DE CHAMADAS DE API RECEBIDAS | 1 MILHÃO DE CHAMADAS DE API HTTP RECEBIDAS | 1 MILHÃO DE MENSAGENS | 750.000 MINUTOS DE CONEXÃO
por mês
Essas ofertas de nível gratuito estão disponíveis somente para novos clientes da AWS pelo período de 12 meses a partir da data de cadastro na AWS. Quando o período de uso gratuito de 12 meses expirar ou se a utilização do aplicativo ultrapassar os níveis de uso gratuito, você simplesmente pagará as tarifas padrão de pagamento conforme o uso.
Cadastre-se no nível gratuito

Figura 9.8: Precificação do AWS API Gateway

Feito o login, você verá uma página semelhante à seguinte:

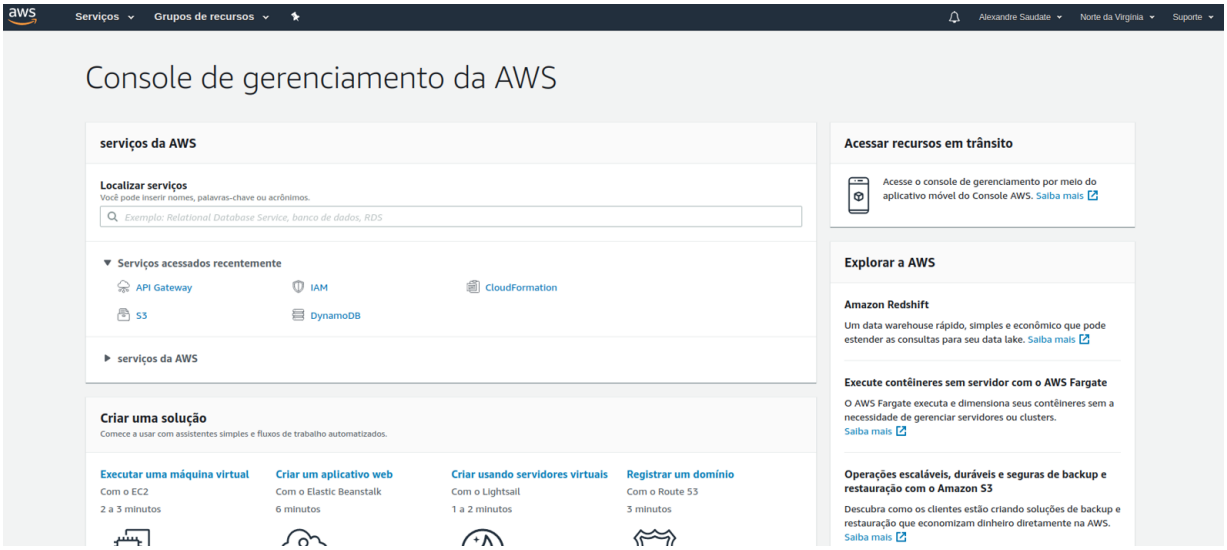

aws
Serviços
Grupos de recursos
Alexandre Saudate
Norte da Virgínia
Suporte
Console de gerenciamento da AWS
serviços da AWS
Localizar serviços
Você pode inserir nomes, palavras-chave ou acrônimos.
Exemplo: Relational Database Service, banco de dados, RDS
Serviços acessados recentemente
API Gateway
IAM
CloudFormation
S3
DynamoDB
serviços da AWS
Criar uma solução
Comece a usar com assistentes simples e fluxos de trabalho automatizados.
Executar uma máquina virtual
Com o EC2
2 a 3 minutos
Criar um aplicativo web
Com o Elastic Beanstalk
6 minutos
Criar usando servidores virtuais
Com o Lightsail
1 a 2 minutos
Registrar um domínio
Com o Route 53
3 minutos
Acessar recursos em trânsito
Acesse o console de gerenciamento por meio do aplicativo móvel do Console AWS. Saiba mais
Explorar a AWS
Amazon Redshift
Um data warehouse rápido, simples e econômico que pode estender as consultas para seu data lake. Saiba mais
Execute contêineres sem servidor com o AWS Fargate
O AWS Fargate executa e dimensiona seus contêineres sem a necessidade de gerenciar servidores ou clusters.
Saiba mais
Operações escaláveis, duráveis e seguras de backup e restauração com o Amazon S3
Descubra como os clientes estão criando soluções de backup e restauração que economizam dinheiro diretamente na AWS.
Saiba mais

Figura 9.9: Página inicial do console da AWS

Digite `API Gateway` na caixa central de busca e dê `enter`. Você estará na página inicial do *serviço* `API Gateway`:

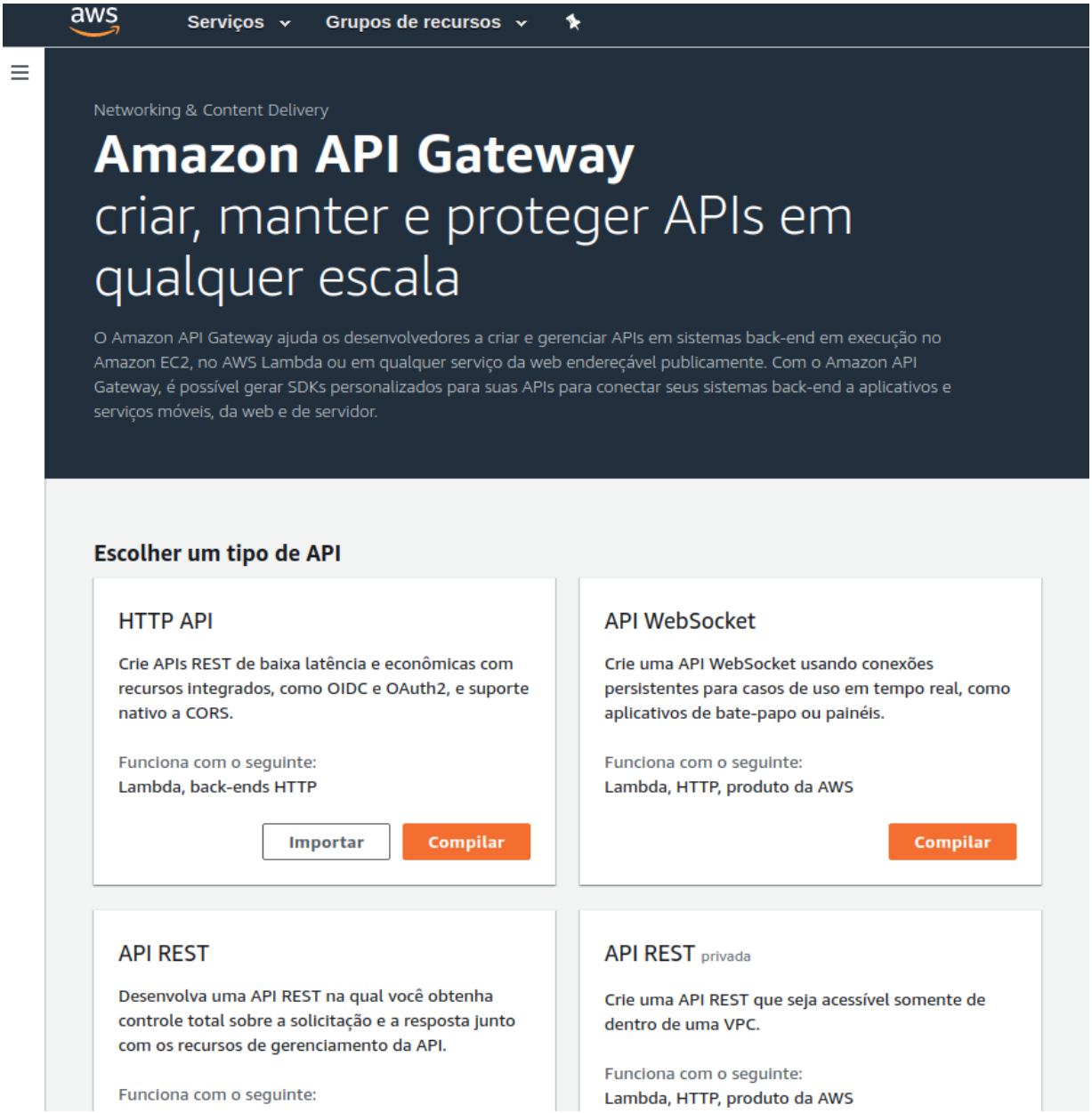

aws
Serviços
Grupos de recursos
Networking & Content Delivery
Amazon API Gateway
criar, manter e proteger APIs em qualquer escala
O Amazon API Gateway ajuda os desenvolvedores a criar e gerenciar APIs em sistemas back-end em execução no Amazon EC2, no AWS Lambda ou em qualquer serviço da web endereçável publicamente. Com o Amazon API Gateway, é possível gerar SDKs personalizados para suas APIs para conectar seus sistemas back-end a aplicativos e serviços móveis, da web e de servidor.
Escolher um tipo de API
HTTP API
Crie APIs REST de baixa latência e econômicas com recursos integrados, como OIDC e OAuth2, e suporte nativo a CORS.
Funciona com o seguinte:
Lambda, back-ends HTTP
Importar
Compilar
API WebSocket
Crie uma API WebSocket usando conexões persistentes para casos de uso em tempo real, como aplicativos de bate-papo ou painéis.
Funciona com o seguinte:
Lambda, HTTP, produto da AWS
Compilar
API REST
Desenvolva uma API REST na qual você obtenha controle total sobre a solicitação e a resposta junto com os recursos de gerenciamento da API.
Funciona com o seguinte:
API REST privada
Crie uma API REST que seja acessível somente de dentro de uma VPC.
Funciona com o seguinte:
Lambda, HTTP, produto da AWS

Figura 9.10: Página principal do AWS API Gateway

Quando clicamos no botão `Compilar` da caixa HTTP API, somos levados a uma tela onde há uma mensagem de boas-vindas:

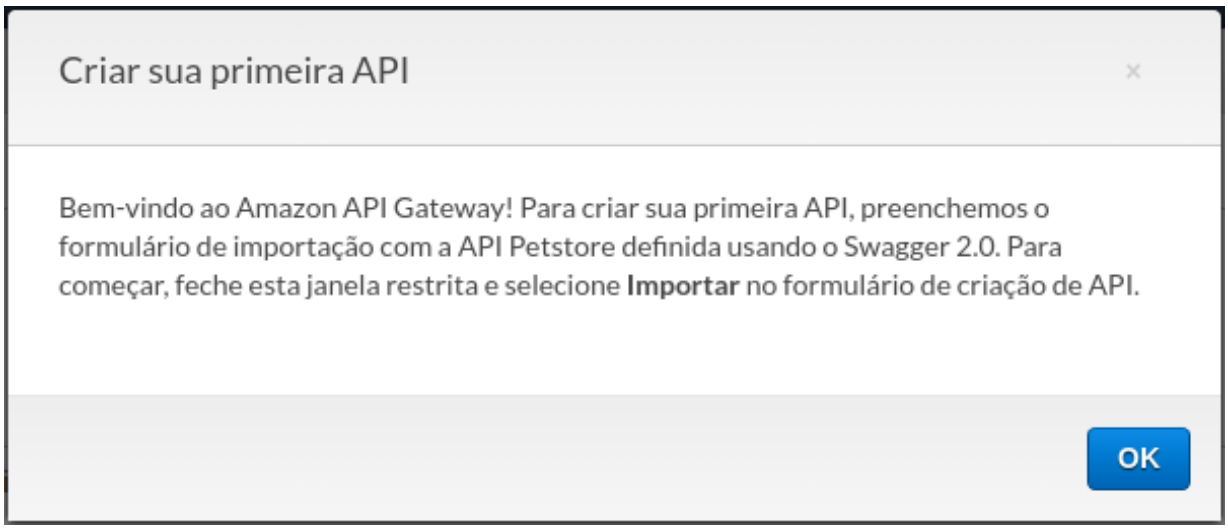

Criar sua primeira API
Bem-vindo ao Amazon API Gateway! Para criar sua primeira API, preenchemos o formulário de importação com a API Petstore definida usando o Swagger 2.0. Para começar, feche esta janela restrita e selecione Importar no formulário de criação de API.
OK

Figura 9.11: Mensagem de boas-vindas do AWS API Gateway

Ao clicar no botão `OK`, você poderá conferir o passo a passo da criação da API, conforme oferecido pela AWS. No momento, existem duas opções disponíveis: via `AWS Lambda` e via `HTTP`. Nosso interesse é via HTTP.

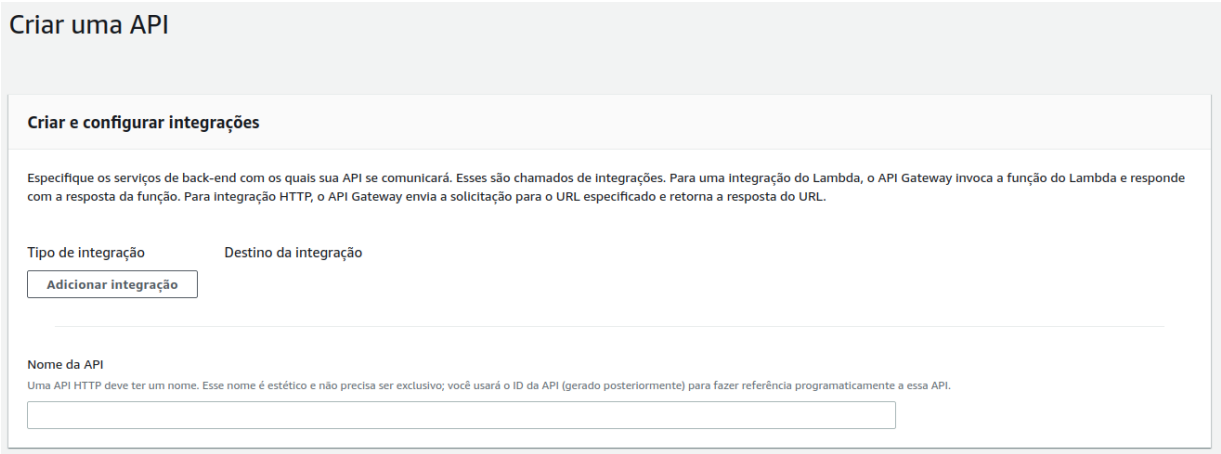

Criar uma API
Criar e configurar integrações
Especifique os serviços de back-end com os quais sua API se comunicará. Esses são chamados de integrações. Para uma integração do Lambda, o API Gateway invoca a função do Lambda e responde com a resposta da função. Para integração HTTP, o API Gateway envia a solicitação para o URL especificado e retorna a resposta do URL.
Tipo de integração
Destino da integração
Adicionar integração
Nome da API
Uma API HTTP deve ter um nome. Esse nome é estético e não precisa ser exclusivo; você usará o ID da API (gerado posteriormente) para fazer referência programaticamente a essa API.

Figura 9.12: Criar API HTTP no AWS API Gateway

Entretanto, temos um problema fundamental: nosso serviço, pelo menos no momento, está disponível apenas na nossa máquina local. É muito complicado realizar a exposição desse serviço em nossa máquina local sem configurações de NAT no roteador (algo que pode ser inviável se você estiver disponibilizando-o na sua

empresa). Existe uma ferramenta que pode nos ajudar nessa tarefa, chamada `ngrok`. O `ngrok` atua exatamente como seu roteador, mas cria um DNS específico para sua máquina. Esse endereço estará disponível na internet e ao alcance de todos - o que permite que uma requisição da AWS chegue diretamente à sua máquina.

Vamos acessar a página do `ngrok` em https://ngrok.com/. Ao clicar no botão de *download*, nos deparamos com uma tela semelhante à seguinte:

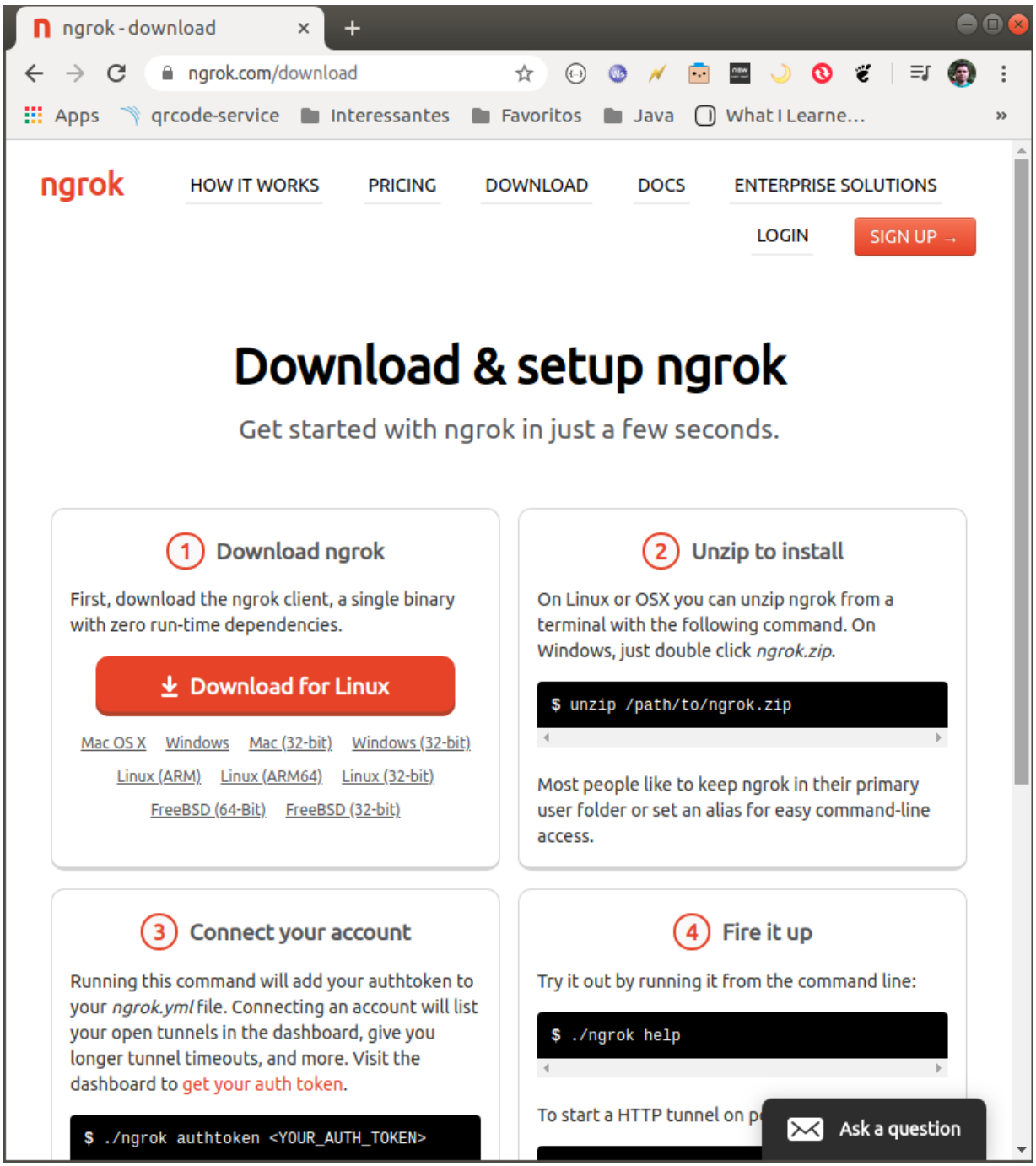

ngrok - download
ngrok.com/download
Apps
qrcode-service
Interessantes
Favoritos
Java
What I Learne...
ngrok
HOW IT WORKS
PRICING
DOWNLOAD
DOCS
ENTERPRISE SOLUTIONS
LOGIN
SIGN UP →
Download & setup ngrok
Get started with ngrok in just a few seconds.
1 Download ngrok
First, download the ngrok client, a single binary with zero run-time dependencies.
Download for Linux
Mac OS X Windows Mac (32-bit) Windows (32-bit)
Linux (ARM) Linux (ARM64) Linux (32-bit)
FreeBSD (64-Bit) FreeBSD (32-bit)
2 Unzip to install
On Linux or OSX you can unzip ngrok from a terminal with the following command. On Windows, just double click ngrok.zip.
$ unzip /path/to/ngrok.zip
Most people like to keep ngrok in their primary user folder or set an alias for easy command-line access.
3 Connect your account
Running this command will add your authtoken to your ngrok.yml file. Connecting an account will list your open tunnels in the dashboard, give you longer tunnel timeouts, and more. Visit the dashboard to get your auth token.
$ ./ngrok authtoken <YOUR_AUTH_TOKEN>
4 Fire it up
Try it out by running it from the command line:
$ ./ngrok help
To start a HTTP tunnel on p
Ask a question

Figura 9.13: Download do Ngrok

Nessa tela, existem as instruções de como baixar e utilizar o `ngrok`, de acordo com seu sistema operacional. Observe que, no meu caso, é um sistema operacional Linux e, portanto, basta baixar, descompactar o zip e utilizar. Não é obrigatório cadastrar o *token*

conforme informado na página, mas algumas *features* só estão disponíveis se você o fizer.

Para disponibilizar nossa API com HTTPS via `ngrok`, o comando é `ngrok http https://localhost:8080`. Isso vai provocar um alerta como o seguinte:

```
asaudate@ni-17935-0:~/Software/ngrok-stable-linux-amd64$ ./ngrok http https://l
ocalhost:8080
Forwarding to local port 443 or a local https:// URL is only available after yo
u sign up.
Sign up at: https://ngrok.com/signup

If you have already signed up, make sure your authtoken is installed.
Your authtoken is available on your dashboard: https://dashboard.ngrok.com

ERR_NGROK_340
```

Figura 9.14: Tentativa de execução do Ngrok com HTTPS

Esbarramos em um novo problema. O `ngrok` com HTTPS só estará disponível se você fizer o registro no site. Se for o caso, basta se autenticar no site do `ngrok` e cadastrar o *token* conforme as instruções. Caso você não queira fazer esse procedimento, podemos remover a camada de segurança da nossa API abrindo o arquivo `application.properties` e colocando `#` na frente das linhas que definem o SSL da nossa aplicação:

```
app.car.domain.googlemaps.apikey=SUA API KEY

#server.ssl.key-store=classpath:keystore.p12
#server.ssl.key-store-password=restbook
#server.ssl.key-store-type=PKCS12
#server.ssl.key-alias=car
```

Vamos então inicializar nossa API novamente e inicializar o `ngrok` com o comando `ngrok http 8080`. Uma tela semelhante à seguinte deve ser vista:

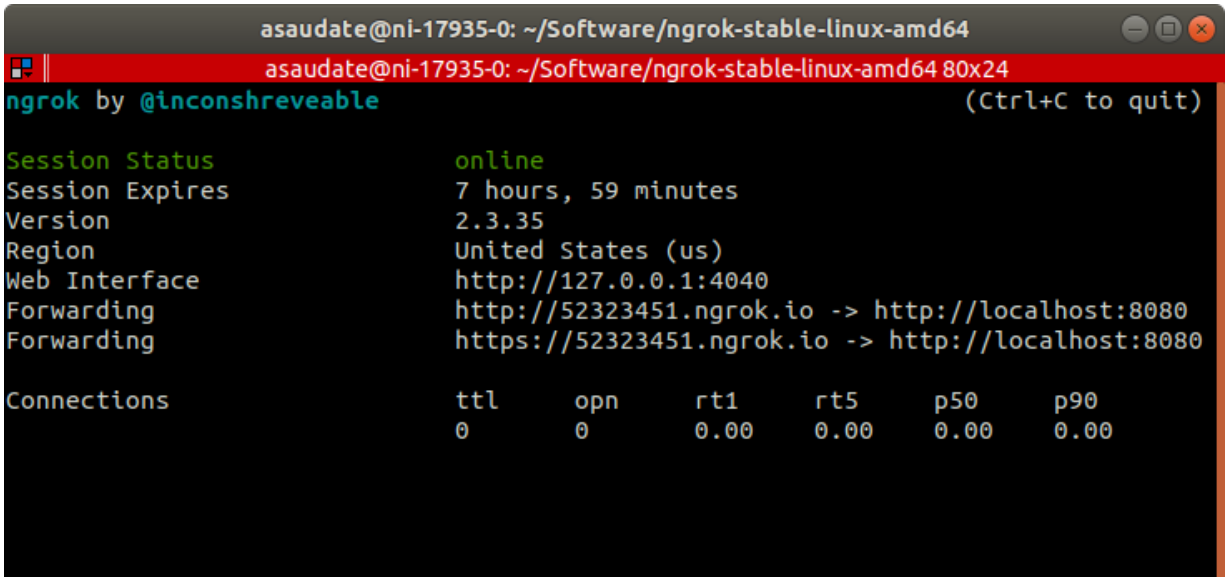

asaudate@ni-17935-0: ~/Software/ngrok-stable-linux-amd64
asaudate@ni-17935-0: ~/Software/ngrok-stable-linux-amd64 80x24
ngrok by @inconshreveable (Ctrl+C to quit)
Session Status online
Session Expires 7 hours, 59 minutes
Version 2.3.35
Region United States (us)
Web Interface http://127.0.0.1:4040
Forwarding http://52323451.ngrok.io -> http://localhost:8080
Forwarding https://52323451.ngrok.io -> http://localhost:8080
Connections ttl opn rt1 rt5 p50 p90
0 0 0.00 0.00 0.00 0.00

Figura 9.15: Ngrok servindo HTTP na porta 8080

Observe que um endereço do tipo `http://xxxxxxxx.ngrok.io` foi criado. Esse endereço muda a cada execução, então temos que tomar o cuidado de realizar os passos seguintes todos de uma só vez.

O próximo passo é voltar ao painel de controle da AWS e adicionar nossa primeira integração. Ao clicar no botão `Adicionar integração`, selecione o campo `HTTP` e, como destino da integração, deixe `ANY` (isso significa que todas as requisições para o mesmo endereço, independente do método HTTP, serão roteadas para o mesmo endereço de destino). Vamos então configurar o endereço conforme a seguir para apontar para a API de motoristas:

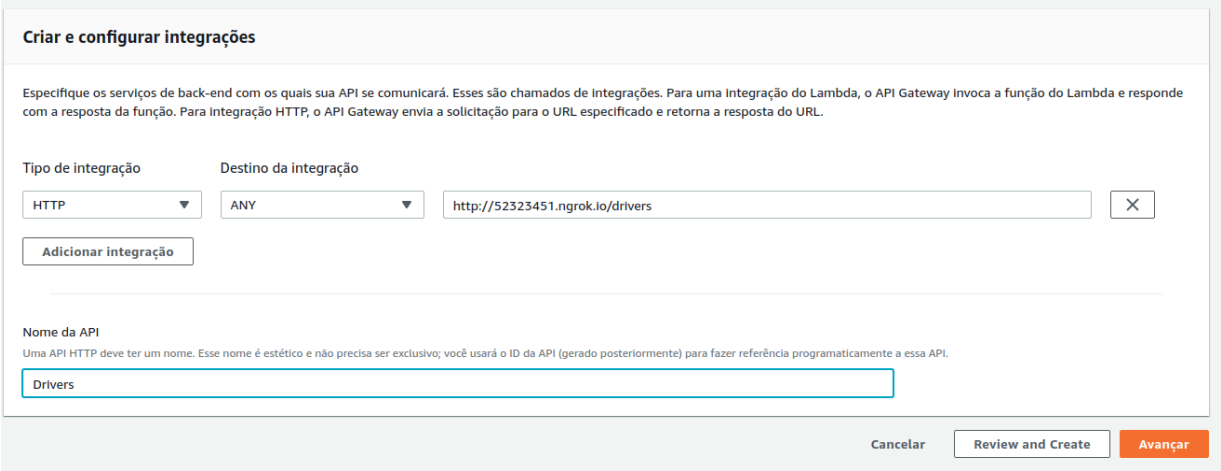

Criar e configurar integrações
Especifique os serviços de back-end com os quais sua API se comunicará. Esses são chamados de integrações. Para uma integração do Lambda, o API Gateway invoca a função do Lambda e responde com a resposta da função. Para integração HTTP, o API Gateway envia a solicitação para o URL especificado e retorna a resposta do URL.
Tipo de integração
Destino da integração
HTTP
ANY
http://52323451.ngrok.io/drivers
Adicionar integração
Nome da API
Uma API HTTP deve ter um nome. Esse nome é estético e não precisa ser exclusivo; você usará o ID da API (gerado posteriormente) para fazer referência programaticamente a essa API.
Drivers
Cancelar
Review and Create
Avançar

Figura 9.16: Adicionando a primeira integração ao API Gateway

Após realizar esse passo, clique no botão `Avançar`. Uma tela de resumo das configurações será apresentada:

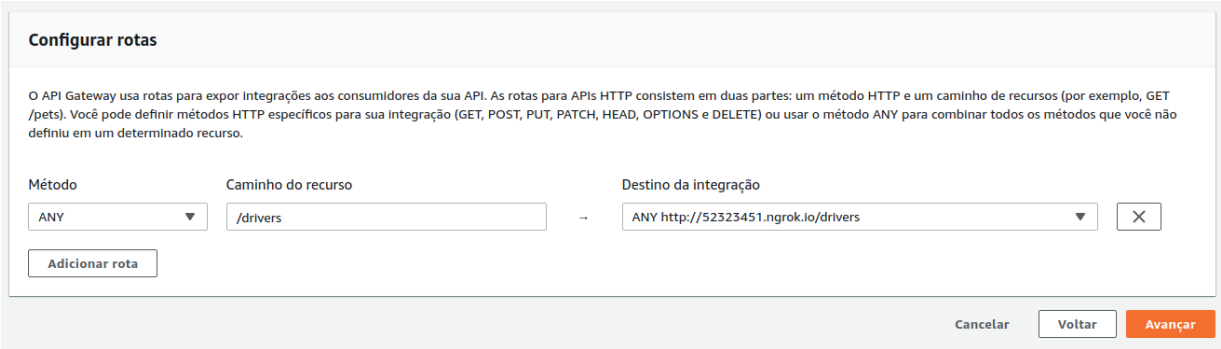

Configurar rotas
O API Gateway usa rotas para expor integrações aos consumidores da sua API. As rotas para APIs HTTP consistem em duas partes: um método HTTP e um caminho de recursos (por exemplo, GET /pets). Você pode definir métodos HTTP específicos para sua integração (GET, POST, PUT, PATCH, HEAD, OPTIONS e DELETE) ou usar o método ANY para combinar todos os métodos que você não definiu em um determinado recurso.
Método
Caminho do recurso
Destino da integração
ANY
/drivers
ANY http://52323451.ngrok.io/drivers
Adicionar rota
Cancelar
Voltar
Avançar

Figura 9.17: Revisando a primeira integração no API Gateway

Quando clicamos em `Avançar`, é apresentada uma nova tela onde a configuração dos **estágios** da API é realizada. Os estágios são semelhantes aos ambientes de desenvolvimento em uma empresa, ou seja, algo como `desenvolvimento`, `homologação`, `sandbox`, `produção` etc. Perceba que, ao entrar na tela, a AWS sugere o nome `$default` como padrão. Vamos alterar para `dev` (diminutivo de desenvolvimento) e deixar marcada a caixa de implantação automática. Isso significa dizer que todas as alterações feitas naquilo que está publicado nesse estágio estarão disponíveis imediatamente para o público, o que é feito para realizar um controle

de aprovação (caso o estágio seja de produção, por exemplo). Em ambiente de desenvolvimento, não precisamos de tanta cautela e, portanto, a caixa pode permanecer marcada:

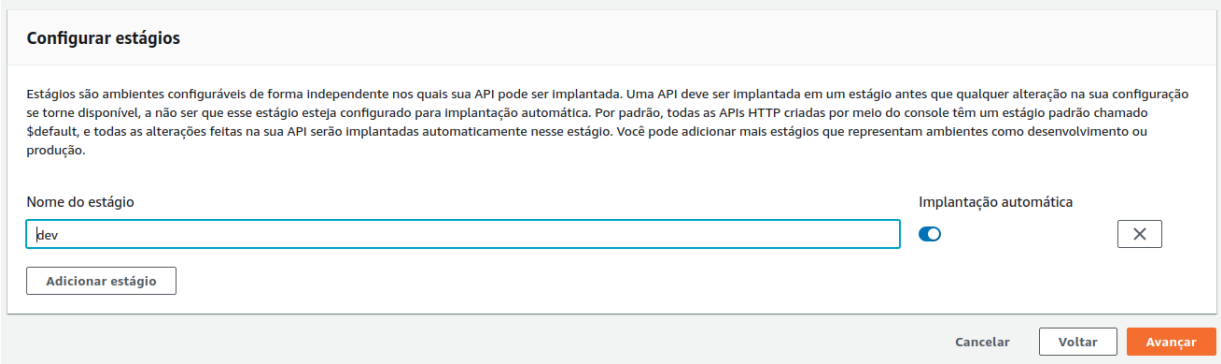

Configurar estágios
Estágios são ambientes configuráveis de forma independente nos quais sua API pode ser implantada. Uma API deve ser implantada em um estágio antes que qualquer alteração na sua configuração se torne disponível, a não ser que esse estágio esteja configurado para implantação automática. Por padrão, todas as APIs HTTP criadas por meio do console têm um estágio padrão chamado $default, e todas as alterações feitas na sua API serão implantadas automaticamente nesse estágio. Você pode adicionar mais estágios que representam ambientes como desenvolvimento ou produção.
Nome do estágio
Implantação automática
dev
Adicionar estágio
Cancelar
Voltar
Avançar

Figura 9.18: Definição de estágios no API Gateway

Finalmente, ao prosseguir com a criação, a AWS nos apresenta uma última tela para revisarmos tudo o que foi criado até aqui:

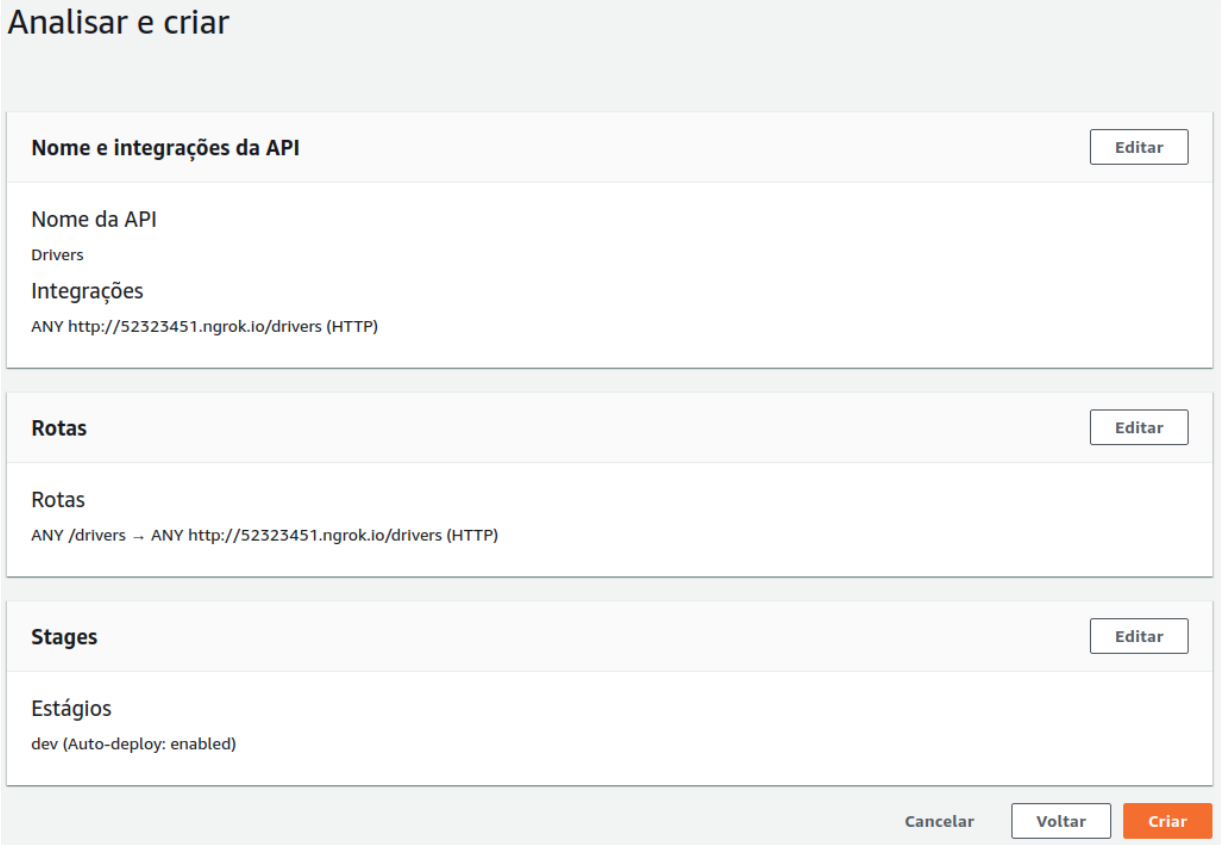


Figura 9.19: Revisão final da configuração do API Gateway

Quando clicamos no botão `Criar`, o API Gateway nos apresenta o painel de controle da API que criamos e, agora, está disponível para realizar a invocação. Observe o campo `Invocar URL`:

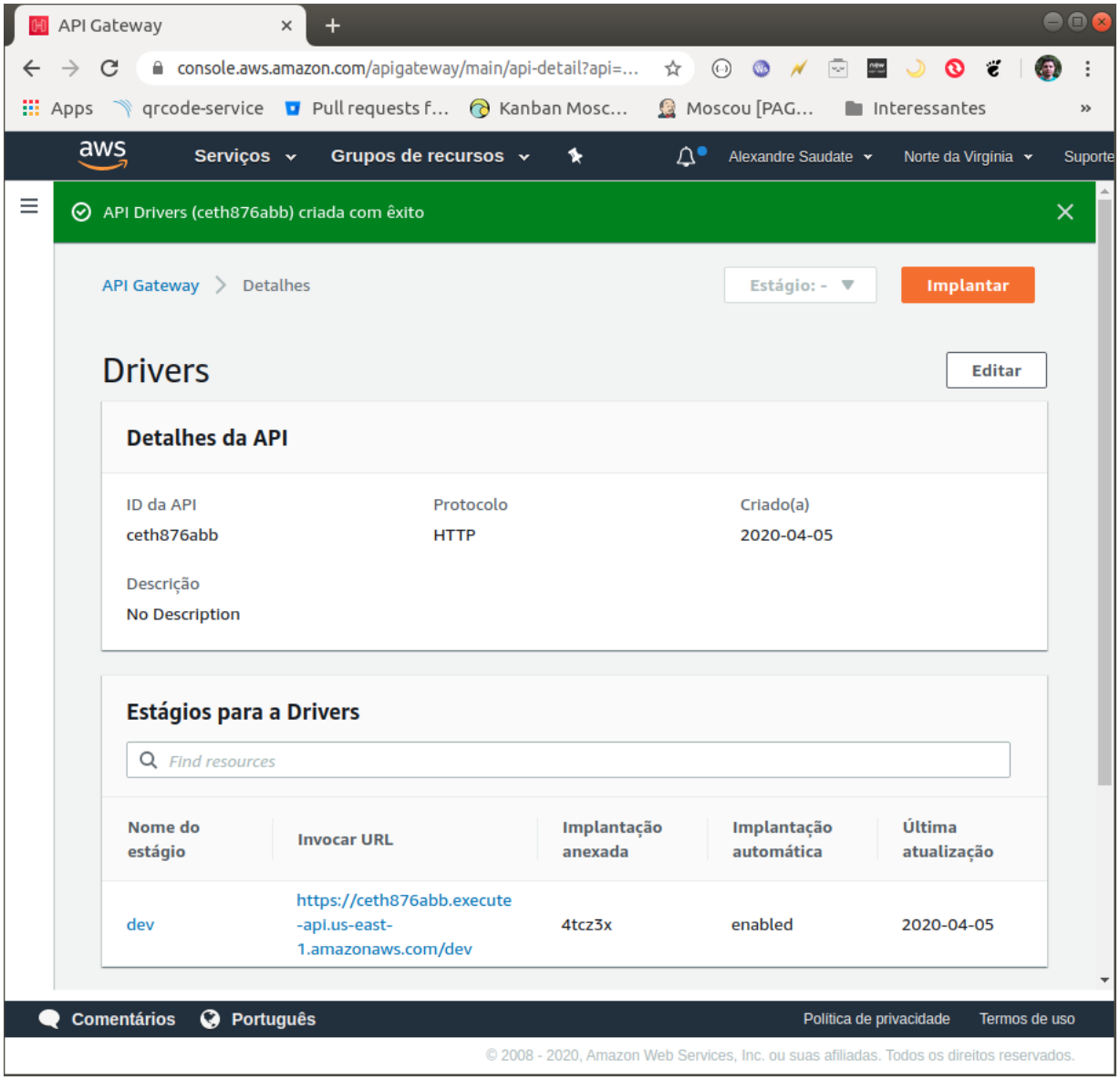


Figura 9.20: API Gateway mostrando o painel de controle da nova API

Segundo este campo, a API está publicada no endereço https://ceth876abb.execute-api-us-east-1.amazonaws.com/dev (no caso deste exemplo específico). Assim sendo, podemos realizar a request para esse endereço, acrescido de `/drivers`. Todo o restante permanece exatamente da mesma forma como era antes, como o corpo da requisição e os cabeçalhos. A requisição ficará assim:

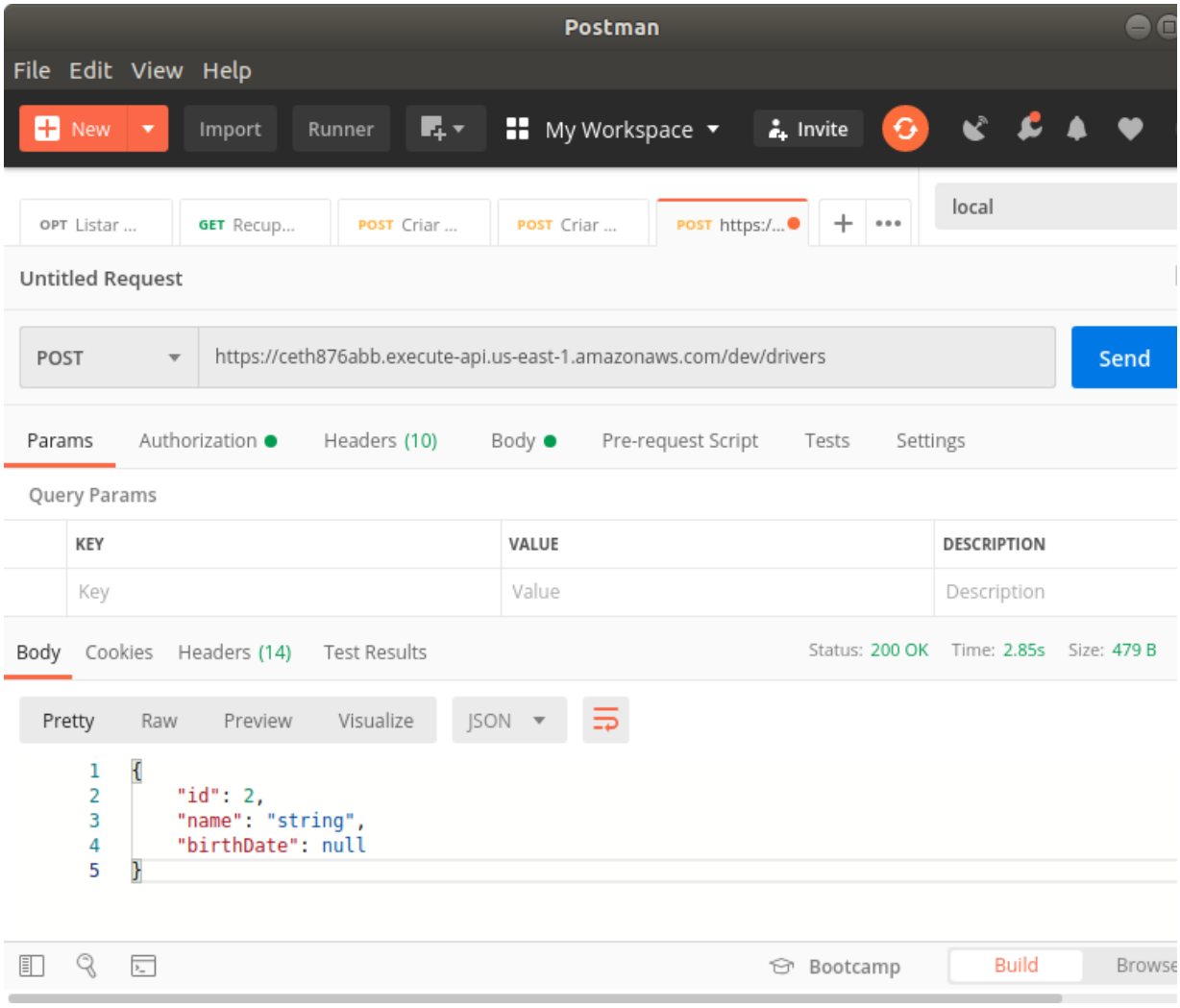


Figura 9.21: Request via Postman ao API Gateway

SE CRIAMOS A API MAPEADA PARA `/drivers`, POR QUE PRECISAMOS INCLUIR NOVAMENTE NA REQUISIÇÃO?

Quando criamos o mapeamento na AWS, inserimos o endereço do `ngrok` e o *path* `/drivers`. Isso porque cada um dos *paths* no API Gateway pode ser mapeado de maneira independente. Se alguma requisição chegar para um *path* que não está mapeado, ela não é entregue. Assim sendo, em realidade não estamos configurando `/drivers` como uma espécie de atalho; estamos apenas solicitando ao API Gateway que ele libere requisições para esse *path*. Quando realizamos a requisição pelo Postman estamos, portanto, tomando proveito desse mapeamento.

Ao utilizar um API Gateway como o da AWS, podemos retirar da aplicação várias preocupações com o uso da API e deixar somente algumas. Podemos deixar de nos preocupar com aspectos, como autenticação/autorização, HTTPS, CORS e outras, pois eles podem ser configurados diretamente no API Gateway, tirando a necessidade de se configurar diretamente na aplicação.

No menu da esquerda do painel, você pode conhecer algumas dessas opções. Um exemplo é o controle de tráfego que será feito. Essa configuração permite que se bloqueie o excesso de tráfego externo à aplicação, poupando a API de uma eventual queda por excesso de requisições.

Para acessar essa configuração, clique no menu à esquerda sobre a opção `Controle de utilização`. Uma vez selecionado o estágio onde a configuração será realizada (no nosso caso, só temos `dev` no momento), um menu como o seguinte será apresentado:

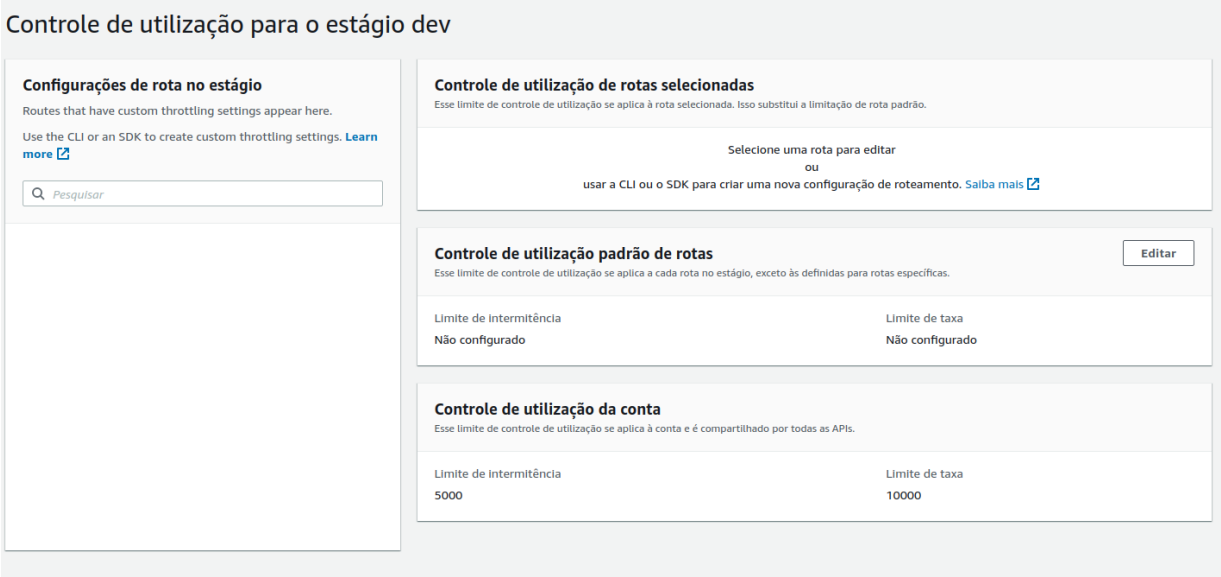

Controle de utilização para o estágio dev
Configurações de rota no estágio
Routes that have custom throttling settings appear here.
Use the CLI or an SDK to create custom throttling settings. Learn more
Pesquisar
Controle de utilização de rotas selecionadas
Esse limite de controle de utilização se aplica à rota selecionada. Isso substitui a limitação de rota padrão.
Selecione uma rota para editar
ou
usar a CLI ou o SDK para criar uma nova configuração de roteamento. Saiba mais
Controle de utilização padrão de rotas
Esse limite de controle de utilização se aplica a cada rota no estágio, exceto às definidas para rotas específicas.
Editar
Limite de intermitência
Não configurado
Limite de taxa
Não configurado
Controle de utilização da conta
Esse limite de controle de utilização se aplica à conta e é compartilhado por todas as APIs.
Limite de intermitência
5000
Limite de taxa
10000

Figura 9.22: Configurações de throttling

No momento, o API Gateway estará com as configurações padrão: até 5.000 requisições simultâneas (no mesmo milissegundo) e até 10.000 requisições no mesmo segundo.

Conclusão

Neste capítulo, conhecemos várias outras técnicas que vão nos ajudar a construir uma API excelente e que vai impulsionar nosso sistema para ser mais e mais embutido em outras aplicações, alavancando o sucesso da nossa própria aplicação.

Nós vimos como criar um bom mecanismo de paginação, garantindo que o retorno das requisições seja leve e fácil de ser utilizado pelos clientes.

Vimos também para que serve e porque configurar CORS - algo muito importante se quisermos que nossa API apresente interoperabilidade com clientes escritos em JavaScript.

Vimos como criar autenticação e autorização utilizando OAuth, um poderoso recurso para quando queremos oferecer nossa aplicação

para ser interoperável não apenas com clientes diretamente, mas também com outras aplicações.

E finalmente, aprendemos a utilizar um recurso poderosíssimo para a efetiva exposição da nossa API para o mundo, o API Gateway. Então, vimos como funciona o API Gateway de um dos principais *players* do mercado, a Amazon Web Services - AWS.

No próximo capítulo, vamos começar a fazer uma revisão geral de tudo que vimos ao longo deste livro para ter certeza de que não deixamos nenhuma informação importante esquecida. Vamos em frente?

Capítulo 10
Considerações finais

"O professor não ensina, mas arranja modos de a própria criança descobrir. Cria situações-problemas." - Jean Piaget.

Existem várias técnicas em APIs REST que eu gostaria de ter abordado neste livro. No entanto, o espaço de que disponho é limitado, então fazer isso não seria possível.

Assim sendo, a seguir enumero vários desses assuntos e como eles podem contribuir para os seus conhecimentos neste universo de APIs.

Cloud computing

Certamente um dos assuntos que estão mais em alta ultimamente. É muito importante que desenvolvedores de APIs detenham o máximo de conhecimento sobre *cloud computing*, já que muitos provedores oferecem ferramentas especializadas para uso nesse tipo de desenvolvimento.

Neste livro, eu abordei o AWS API Gateway. Observe que a AWS oferece várias ferramentas que estão interligadas com esse universo, como o Route 53 (para administrar DNS) e CloudFront (para criação de *content delivery networks*, ou `CDNs`), além de diversas opções de implantação de APIs em servidores gerenciados. Da mesma forma, concorrentes da AWS, como Google Cloud Platform e Microsoft Azure oferecem muitas ferramentas nesse sentido.

Serverless

Ao abordar API Gateways, confesso a você que fiquei tentado a me debruçar, ainda neste livro, sobre o tema *serverless*. Trata-se de uma forma de criar e executar uma aplicação em nuvem sem ter

que lidar com o fardo de criar e gerenciar vários aspectos de servidores gerenciados - a própria nuvem faz isso por nós. Quando utilizamos *serverless* (do inglês, **sem servidor**) precisamos nos preocupar apenas com o código em si e na forma como ele está exposto para o mundo, e não com detalhes da execução do código. Isso faz com que essa técnica seja extremamente poderosa e **barata**, já que o provedor de nuvem aloca processamento para o código apenas quando é estritamente necessário, derrubando os custos de manter um servidor executando o tempo inteiro.

Não bastassem esses aspectos, APIs *serverless* hoje são as mais escaláveis conhecidas, já que, como o processamento é gerenciado diretamente pelo *cloud provider* que estamos utilizando, a escalabilidade delas tende a ser o tanto quanto o provedor puder disponibilizar. A AWS, hoje, é responsável por grande parte do tráfego mundial de internet.

CQRS

O padrão CQRS, *Command Query Responsibility Segregation*, é uma alternativa às APIs REST. Em outras palavras, uma API CQRS não pode ser considerada RESTful, pois não tem as mesmas características de um serviço REST. CQRS tem como missão lidar com questões que são mais complexas de lidar em APIs RESTful, como buscas com parâmetros complexos e cenários onde a criação de recursos e a efetiva localização destes é destoante. O CQRS tem a missão de separar os aspectos de criação/atualização dos aspectos de localização dos recursos.

GraphQL

Uma técnica que também não é RESTful, mas tem a missão de criar uma API na qual seja possível criar buscas de forma dinâmica, sem que seja necessário deixá-las previamente criadas no lado do servidor. Ao utilizar GraphQL, o cliente pode submeter a sua *query* customizada e pesquisar informações de acordo com sua necessidade - semelhante ao que ocorre conosco quando utilizamos

um banco de dados relacional, por exemplo. Para saber mais sobre GraphQL, visite o site https://graphql.org/. Você também pode conhecer o livro https://www.casadocodigo.com.br/products/livro-graphql.

OAuth

No capítulo 9, eu abordei OAuth de maneira superficial, apenas por considerar o quanto é importante o conhecimento sobre essa técnica. Felizmente, existe um livro na Casa do Código inteiramente dedicado ao assunto, disponível em https://www.casadocodigo.com.br/products/livro-oauth. É um ótimo ponto de partida para que você possa se aprofundar mais sobre o assunto.

JWT

Ainda no quesito "segurança de APIs", temos o JWT - JSON Web Token. Trata-se de um padrão criado para que o próprio sistema de segurança da API consiga trafegar mais dados sobre o usuário do que tão somente o nome do usuário e senha. Isso habilita cenários complexos de autenticação e autorização envolvendo múltiplos participantes, tornando o conhecimento em JWT algo crucial para o desenvolvimento de APIs públicas e interconectadas. Para saber mais, visite o site https://jwt.io/.

Reactive APIs

As APIs reativas oferecem uma forma moderna de lidar com concorrência em ambientes que são requisitados por muitos usuários ao mesmo tempo. Elas fazem um uso melhor de I/O não blocante, o que por sua vez as leva a utilizarem de maneira mais eficiente recursos externos. Assim sendo, são extremamente úteis em contextos de múltiplos microsserviços.

gRPC

gRPC é uma forma de criar APIs RPC (*Remote Procedure Call*, ou Chamada remota de procedimento). Em oposição a REST, onde a ação é executada no servidor de forma indireta, em RPC o cliente determina exatamente qual é a ação a ser invocada. Por exemplo, uma chamada RPC hipotética poderia ter o seguinte formato:

```
POST /actions

{
 "acao":"criarPassageiro",
 "dados": "..."
}
```

O gRPC é uma abordagem para APIs RPC que nasceu no Google e hoje é incubado em um projeto chamado *Cloud Native Computing Foundation*. O gRPC visa tornar comunicações RPC mais eficientes por meio de uma ferramenta de serialização nascida no Google, a *Protocol Buffers*. Através dessa ferramenta, os formatos de dados são pré-compilados, o que viabiliza a serialização mais rápida de dados.

Em linhas gerais, o gRPC se opõe a REST em quase todos os aspectos. Uma arquitetura de APIs eficiente poderia se beneficiar de ambos os formatos - oferecer mais opções de uso para os clientes pode ser bastante benéfico para todos.

Para saber mais sobre o gRPC, visite o site https://grpc.io/.

Capítulo 11
Referências bibliográficas

EVANS, Eric. *Domain-driven design:* Tackling complexity in the heart of software. Addison-Wesley Professional, 2003.

FIELDING, Roy Thomas. *Architectural styles and the design of networkbased software architectures.* 2000. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

FOWLER, Martin. *FluentInterface*. 2005. Disponível em: https://martinfowler.com/bliki/FluentInterface.html.

HAMMER, Eran. *Introducing OAuth 2.0*. 2010. Disponível em: https://hueniverse.com/introducing-oauth-2-0-b5681da60ce2.

KELLY, Mike. *HAL - Hypertext Application Language*. 2013. Disponível em: http://stateless.co/hal_specification.html.

LEWIS, James; FOWLER, Martin. *Microsserviços em poucas palavras*. 2015. Disponível em: https://www.thoughtworks.com/pt/insights/blog/microservices-nutshell.

MARTIN, Robert C. *The Clean Architecture*. 2012. Disponível em: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

WACKER, Mike. *Just Say No To More End-To-End Tests*. 2015. Disponível em: https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html.

WIGGINS, Adam. *The Twelve-Factor App*. 2017. Disponível em: https://12factor.net/pt_br/.